Basics of Statistical Learning

David Dalpiaz

2019-11-18

Contents

I A Machine Learning Preview **11**

1 Introduction **13**

2 Regression: Powerlifting **15**
2.1 Background . . . 15
2.2 Data . . . 16
2.3 EDA . . . 16
2.4 Modeling . . . 20
2.5 Model Evaluation . . . 21
2.6 Discussion . . . 23

3 Classification: Handwriting **25**
3.1 Background . . . 25
3.2 Data . . . 26
3.3 EDA . . . 26
3.4 Modeling . . . 27
3.5 Model Evaluation . . . 27
3.6 Discussion . . . 28

4 Clustering: Basketball Players **31**
4.1 Background . . . 32
4.2 Data . . . 32
4.3 EDA . . . 33
4.4 Modeling . . . 35
4.5 Model Evaluation . . . 37
4.6 Discussion . . . 39

II Some Machine Learning Foundations **41**

5 Introduction **43**

6 Probability **45**
6.1 Probability Models . . . 45

6.2 Probability Axioms . . . 46
6.3 Probability Rules . . . 46
6.4 Random Variables . . . 48
6.4.1 Distributions . . . 48
6.4.2 Discrete Random Variables . . . 48
6.4.3 Continuous Random Variables . . . 49
6.4.4 Several Random Variables . . . 50
6.5 Expectations . . . 50
6.6 Likelihood . . . 51
6.7 Videos . . . 51
6.8 References . . . 52

7 Estimation **53**
7.1 Probability . . . 53
7.2 Statistics . . . 53
7.3 Estimators . . . 53
7.3.1 Properties . . . 54
7.3.2 Methods . . . 54

III A Tour of Machine Learning **57**

8 Introduction **59**

9 Regression **61**
9.1 Setup . . . 62
9.2 Modeling . . . 63
9.2.1 Linear Models . . . 63
9.2.2 k-Nearest Neighbors . . . 64
9.2.3 Decision Trees . . . 66
9.3 Procedure . . . 68
9.4 Data Splitting . . . 68
9.5 Metrics . . . 68
9.6 Model Complexity . . . 69
9.7 Overfitting . . . 69
9.8 Multiple Features . . . 70
9.9 Example Analysis . . . 70
9.10 MISC TODOS . . . 70

10 Bias–Variance Tradeoff **73**
10.1 Reducible and Irreducible Error . . . 74
10.2 Bias-Variance Decomposition . . . 75
10.3 Simulation . . . 78
10.4 Estimating Expected Prediction Error . . . 87
10.5 Reproducibility . . . 88

11 Classification **89**
11.1 STAT 432 Materials . . . 89
11.2 Bayes Classifier . . . 89
11.2.1 Bayes Error Rate . . . 90
11.3 Building a Classifier . . . 90
11.4 Modeling . . . 92
11.4.1 Linear Models . . . 92
11.4.2 k-Nearest Neighbors . . . 92
11.4.3 Decision Trees . . . 92
11.5 MISC TODO STUFF . . . 92

12 Resampling **95**
12.1 STAT 432 Materials . . . 95
12.2 Validation-Set Approach . . . 97
12.3 Cross-Validation . . . 98
12.4 Test Data . . . 101
12.5 MISC TODOS . . . 106

13 Supervised Learning **107**

14 Regularization **109**
14.1 STAT 432 Materials . . . 109
14.2 Reducing Variance with Added Bias . . . 109
14.3 scaling matters? . . . 111
14.4 Constraints in Two Dimensions . . . 111
14.5 High Dimensional Data . . . 113
14.6 Ridge Regression . . . 114
14.7 Lasso . . . 117
14.8 boston is boring . . . 117
14.9 some more simulation . . . 122

15 Ensembles **123**
15.1 STAT 432 Materials . . . 123
15.2 Bagging . . . 123
15.2.1 Simultation Study . . . 126
15.3 Random Forest . . . 126
15.4 Boosting . . . 132

16 Practical Issues **139**
16.1 STAT 432 Materials . . . 139
16.2 Feature Scaling . . . 139
16.3 Categorical Features . . . 144

IV Mathematics **149**

17 Introduction **151**

V Computing **153**

18 Introduction **155**

19 Resources **157**
19.1 Resources . . . 157
19.1.1 R . . . 158
19.1.2 RStudio . . . 158
19.1.3 R Markdown . . . 158
19.2 BSL Idioms . . . 158
19.2.1 Reference Style . . . 158
19.2.2 BSL Style Overrides . . . 159
19.2.3 Objects and Functions . . . 159
19.2.4 Print versus Return . . . 160
19.2.5 Help . . . 161
19.2.6 Keyboard Shortcuts . . . 162
19.3 Common Issues . . . 162

20 Simulation and Bootstrap **163**
20.1 STAT 432 Materials . . . 163
20.2 Misc Notes . . . 172

21 Data Manipulation **173**
21.1 `dplyr` . . . 173
21.2 `data.table` . . . 173
21.3 Data Splitting with `dplyr::anti_join` . . . 173

VI Analysis **175**

22 Introduction **177**

VII Appendix **179**

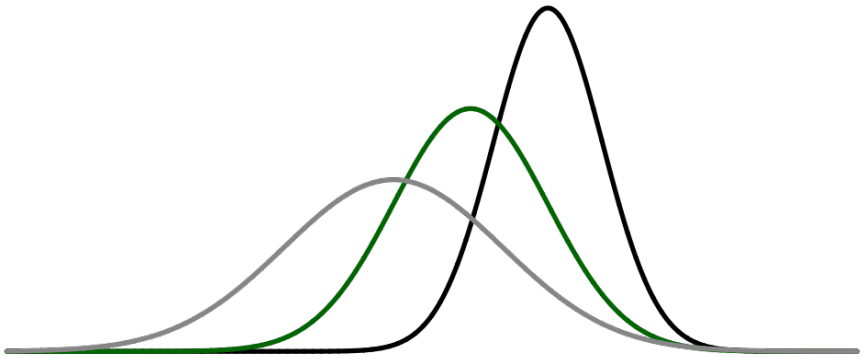
23 Introduction **181**

24 Additional Reading **183**
24.1 Books . . . 183
24.2 Papers . . . 184
24.3 Blog Posts . . . 184
24.4 Miscellaneous . . . 184

25 Class Imbalance **185**

26 Missing Data **193**

Preface



Welcome to Basics of Statistical Learning! What a boring title! The title was chosen to mirror the University of Illinois course STAT 432 - Basics of Statistical Learning. That title was chosen to meet certain University course naming conventions, hence the boring title. A more appropriate title would be "Machine Learning from the Perspective of a Statistician who uses R," which is more descriptive, but still a boring title. Anyway, this book will often be referred to as **BSL**.[1]

Caveat Emptor

This "book" is under active development. Literally every element of the book is subject to change, at any moment. This text, BSL, is the successor to R4SL, an unfinished work that began as a supplement to Introduction to Statistical Learning, but was never finished. (In some sense, this book is just a

[1] Just an example footnote, please ignore.

fresh start due to the author wanting to change the presentation of the material. The author is seriously worried that he will encounter the second-system effect.)

Because this book is written with a course in mind, that is actively being taught, sometimes out of convenience, the text will speak directly to the students of that course. Thus, be aware that any reference to a "course" are a reference to STAT 432 @ UIUC.

A **PDF** version is maintained for use offline, however, given the pace of development, this should only be used if absolutely necessary. During development formatting in the PDF version will largely be ignored.

Since this book is under active development you may encounter errors ranging from typos, to broken code, to poorly explained topics. If you do, please let us know! Better yet, fix the issue yourself! If you are familiar with R Markdown and GitHub, pull requests are highly encouraged!. This process is partially automated by the edit button in the top-left corner of the html version. If your suggestion or fix becomes part of the book, you will be added to the list at the end of this chapter. We'll also link to your GitHub account, or personal website upon request. If you're not familiar with version control systems feel free to email the author, `dalpiaz2 AT illinois DOT edu`. (But also consider using this opportunity to learn a bit about version control!) See additional details in the Acknowledgements section.

While development is taking place, you may see "TODO" scattered throughout the text. These are mostly notes for internal use, but give the reader some idea of what development is still to come. For additional details on the development process, please see the README file on GitHub as well as the Issues page.

Who?

This book is targeted at advanced undergraduate or first year MS students in Statistics who have no prior machine learning experience. While both will be discussed in great detail, previous experience with both statistical modeling and R are assumed.

Organization

Note: This is somewhat speculative.

01 - A Machine Learning Preview

1. Introduction
2. Regression (Powerlifting)
3. Classification (Handwriting)
4. Clustering (NBA Players)

02 - Some Machine Learning Foundations

1. Introduction
2. Probability (A Quick Tour)
3. Statistics and Estimation (A Quick Tour)
4. Density Estimation (?)

03 - A Tour of Machine Learning

1. Introduction
    - Data Splitting
    - Generalization to Unseen Data
2. Regression
3. Bias, Variane, Loss, Risk
4. Classification
5. Resampling
6. Recap and Overview of Supervised Learning
7. Regularization
8. Ensemble Learning
9. Practical Issues
10. Clustering (Unsupervised Learning)
    - Recap the unsupervised learning that was done throughout.
    - Introduce clustering specifically.

04 - Mathematics (Miscellaneous chapters further exploring mathematical details)

1. Introduction
2. Bayes Theorem
3. Multivariate Normal

05 - Computing (Miscellaneous chapters further exploring computing details)

1. Introduction
2. Getting up to speed with R (Currently "Computing" in the main narrative.)
3. `purrr::map()`
4. Simulation? Bootstrap?

06 - Analysis (Examples of using the material with "real world" datasets)

07 - Appendix (Additional Readings and References)

- Misc papers, blogs, tutorials, etc
- Misc videos

Figure 1: CC NC SA

Acknowledgements

The following is a (likely incomplete) list of helpful contributors. This book was also influenced by the helpful contributors to R4SL.

- Jae-Ho Lee - STAT 432, Fall 2019

Your name could be here! Please see the CONTRIBUTING document on GitHub for details on interacting with this project. Pull requests encouraged!

Looking for ways to contribute?

- You'll notice that a lot of the plotting code is not displayed in the text, but is available in the source. Currently that code was written to accomplish a task, but without much thought about the best way to accomplish the task. Try refactoring some of this code.
- Fix typos. Since the book is actively being developed, typos are getting added all the time.
- Suggest edits. Good feedback can be just as helpful as actually contributing code changes.

TODO: Standing on the shoulder of giants. High level acknowledgements.

License

Part I

A Machine Learning Preview

Chapter 1

Introduction

- TODO: The Pokemon introduction?
  - Regression
  - Classification
  - Clustering

Chapter 2

Regression: Powerlifting

```
library(readr)
library(tibble)
library(dplyr)
library(purrr)
library(ggplot2)
library(ggridges)
library(lubridate)
library(randomForest)
library(rpart)
library(rpart.plot)
library(cluster)
library(caret)
library(factoextra)
library(rsample)
library(janitor)
library(rvest)
library(dendextend)
library(knitr)
library(kableExtra)
library(ggthemes)
```

- TODO: Show package messaging? check conflicts!
- TODO: Should this be split into three analyses with different packages?

2.1 Background

- TODO: https://www.openpowerlifting.org/

- TODO: https://en.wikipedia.org/wiki/Powerlifting

2.2 Data

- TODO: Why readr::col_factor() and not just col_factor()?
- TODO: Characters should be character and "categories" should be factors.
- TODO: Is `na.omit()` actually a good idea?

```
pl = read_csv("data/pl.csv", col_types = cols(Sex = readr::col_factor()))
```

```
pl
```

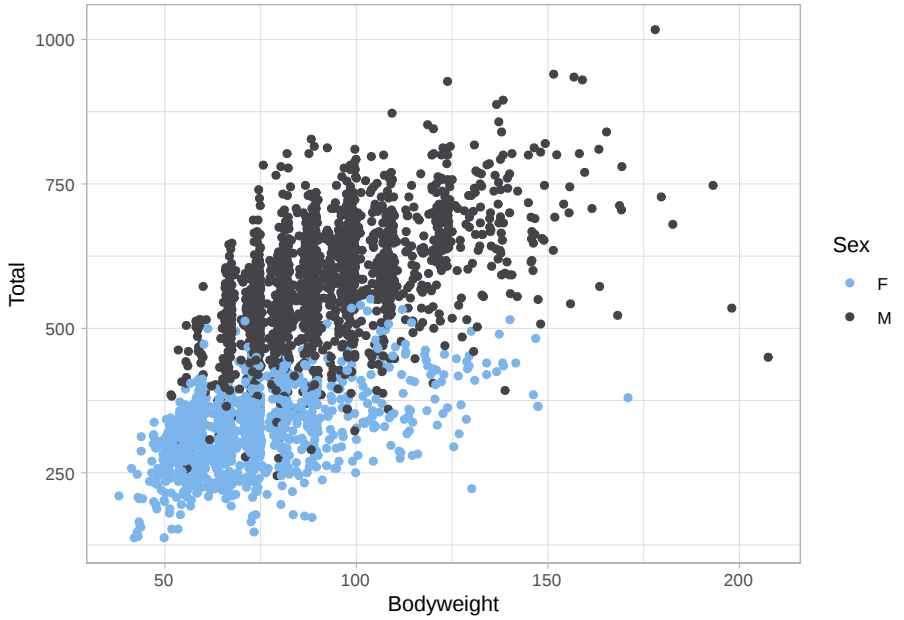
```
## # A tibble: 3,604 x 8
##    Name             Sex   Bodyweight   Age Squat Bench Deadlift Total
##    <chr>            <fct>      <dbl> <dbl> <dbl> <dbl>    <dbl> <dbl>
##  1 Ariel Stier      F             60    32 128.   72.5     150   350
##  2 Nicole Bueno     F             60    26 110    60       135   305
##  3 Lisa Peterson    F             67.5  28 118.   67.5     138.  322.
##  4 Shelby Bandula   F             67.5  26  92.5  67.5     140   300
##  5 Lisa Lindhorst   F             67.5  28  92.5  62.5     132.  288.
##  6 Laura Burnett    F             67.5  30  90    45       108.  242.
##  7 Suzette Bradley  F             75    38 125    75       158.  358.
##  8 Norma Romero     F             75    20  92.5  42.5     125   260
##  9 Georgia Andrews  F             82.5  29 108.   52.5     120   280
## 10 Christal Bundang F             90    30 100    55       125   280
## # ... with 3,594 more rows
```

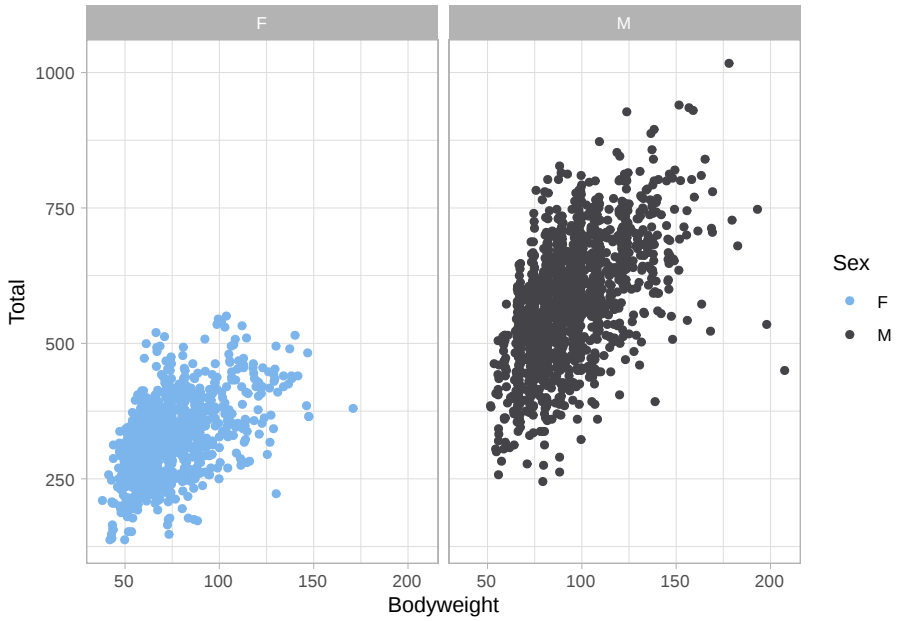
2.3 EDA

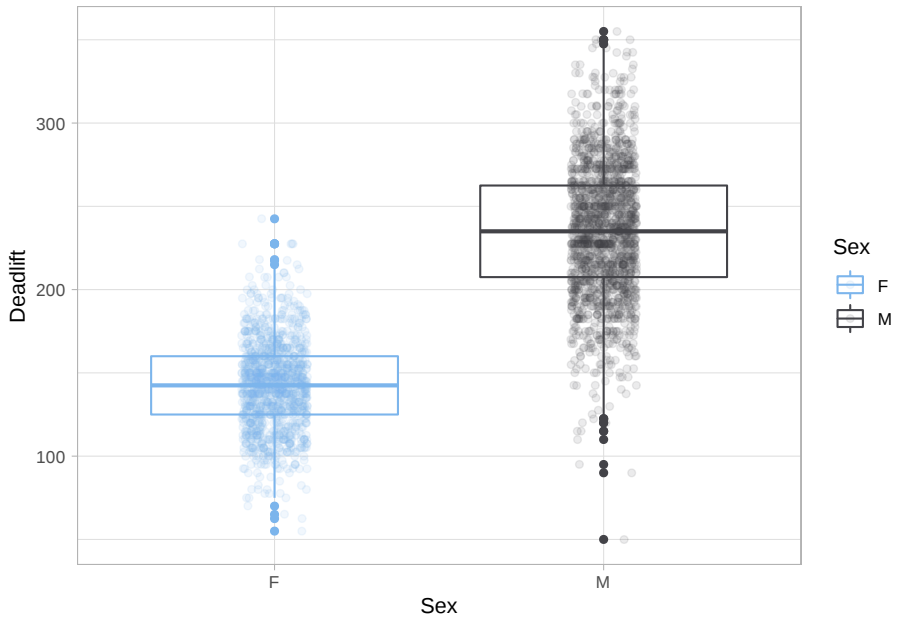
```
set.seed(1)

# test-train split
pl_tst_trn_split = initial_split(pl, prop = 0.80)
pl_trn = training(pl_tst_trn_split)
pl_tst = testing(pl_tst_trn_split)

# estimation-validation split
pl_est_val_split = initial_split(pl_trn, prop = 0.80)
pl_est = training(pl_est_val_split)
pl_val = testing(pl_est_val_split)
```

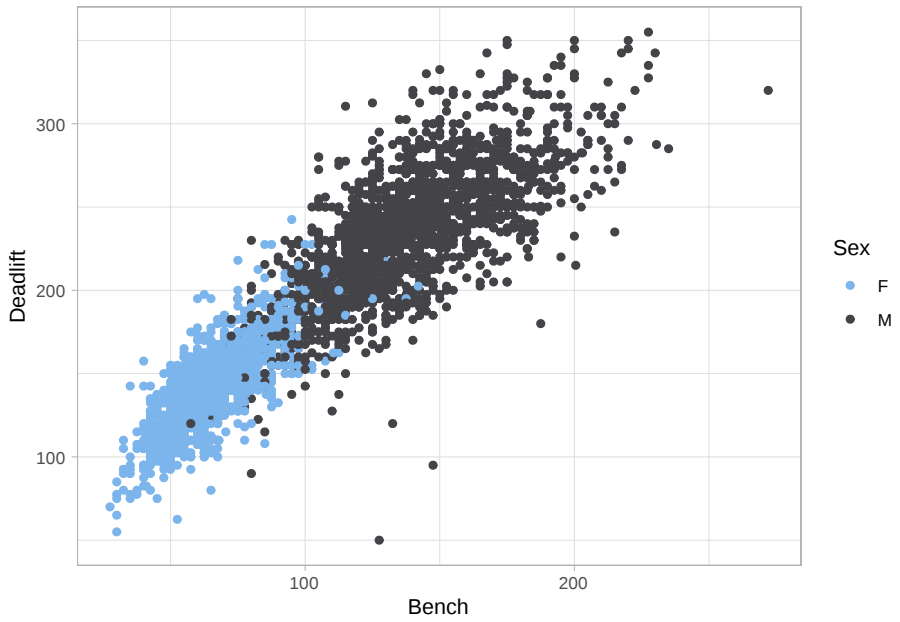
```
rm(pl)
```

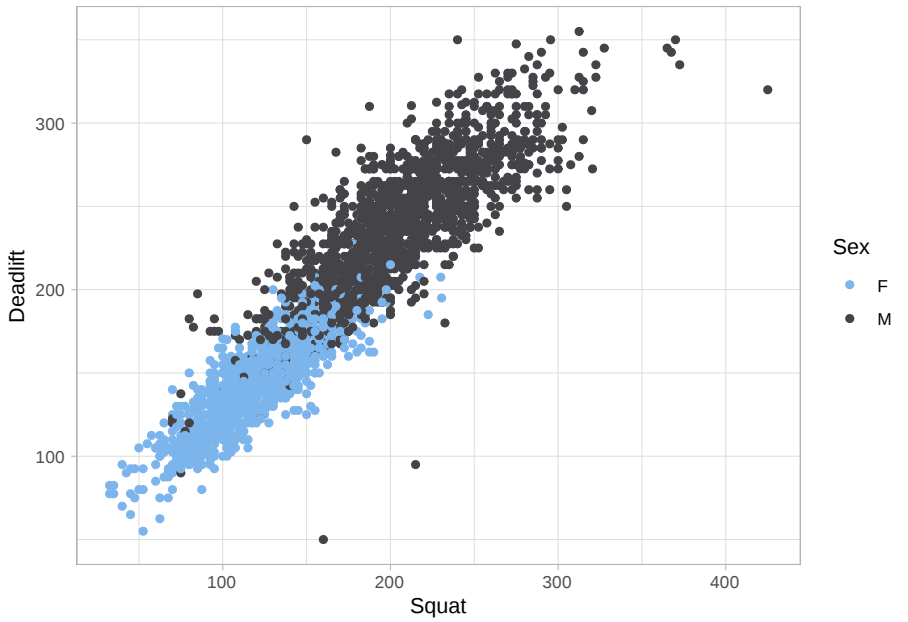
- TODO: Train can be used however you want. (Including EDA.)
- TODO: Test can only be used after all model decisions have been made!






300
200
100
Deadlift
F
M
Sex
Sex
F
M

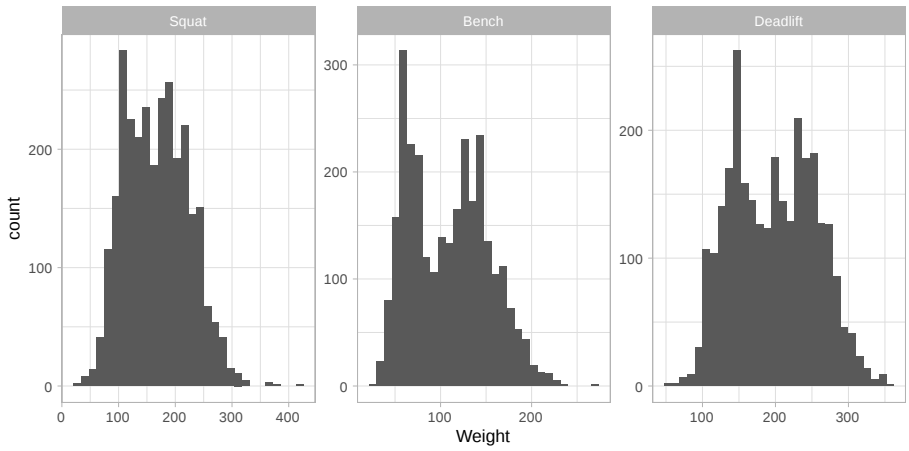

300
200
100
Deadlift
100
200
Bench
Sex
F
M

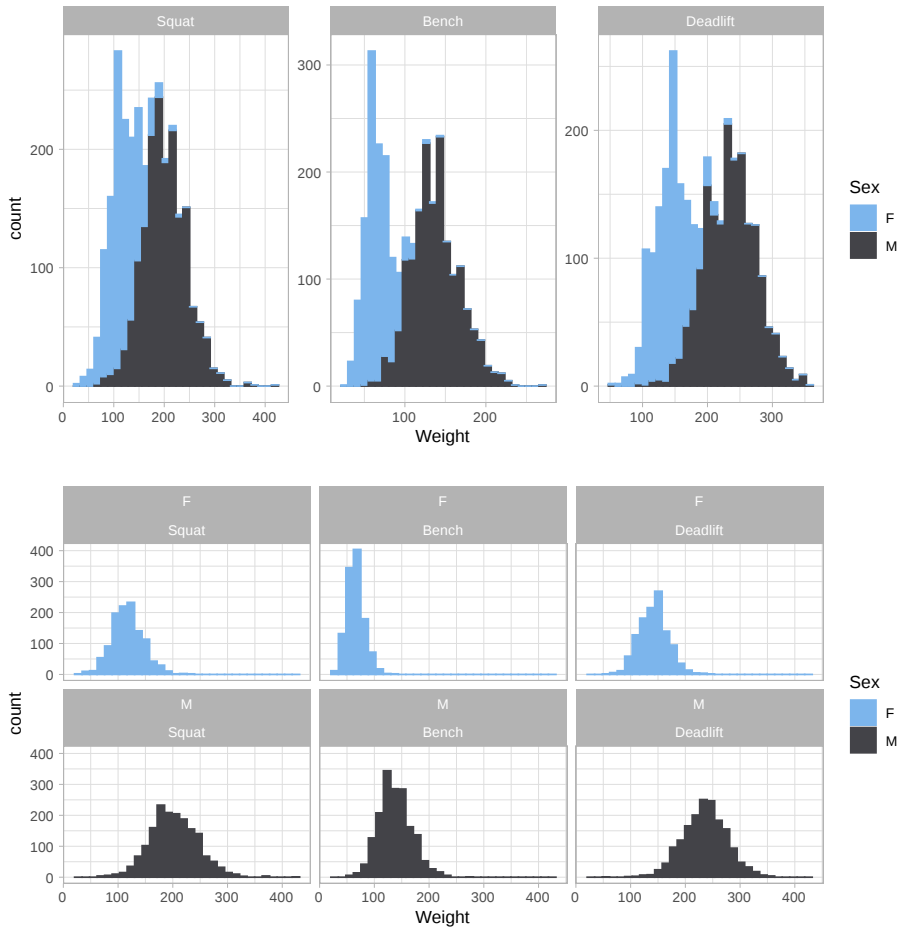



```
pl_trn_tidy = gather(pl_trn, key = "Lift", value = "Weight",
                     Squat, Bench, Deadlift)
```

```
pl_trn_tidy$Lift = factor(pl_trn_tidy$Lift, levels = c("Squat", "Bench", "Deadlift"))
```

- TODO: https://www.tidyverse.org/
- TODO: https://en.wikipedia.org/wiki/Tidy_data
- TODO: http://vita.had.co.nz/papers/tidy-data.pdf





2.4 Modeling

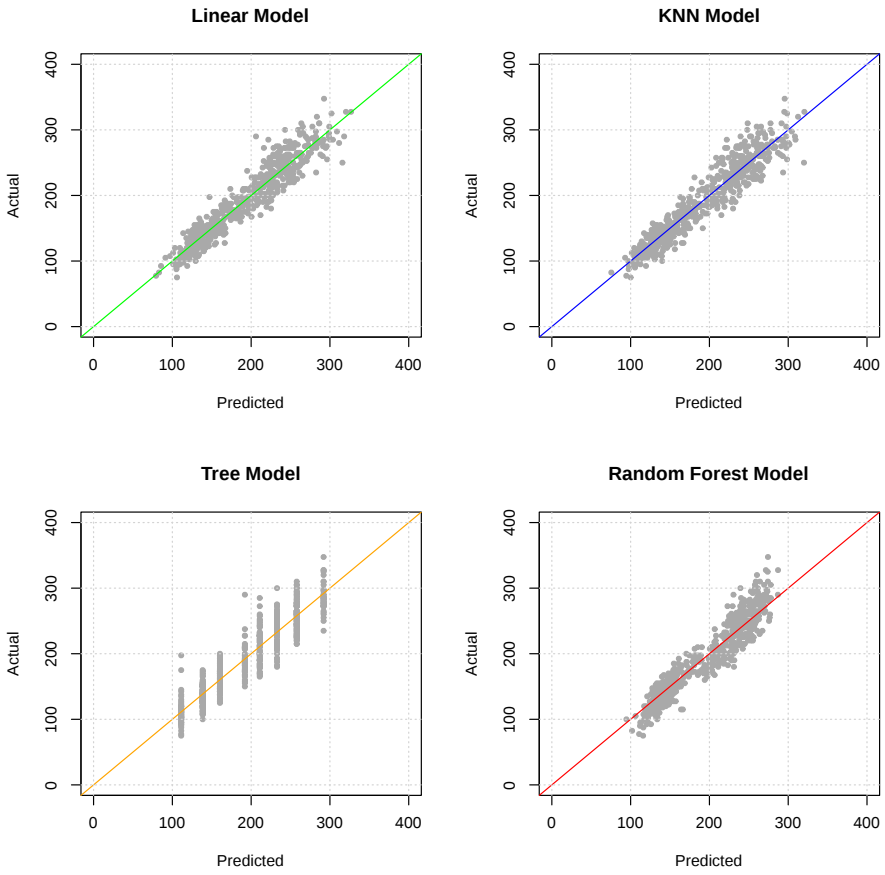
```
dl_mod_form = formula(Deadlift ~ Sex + Bodyweight + Age + Squat + Bench)

set.seed(1)
lm_mod  = lm(dl_mod_form, data = pl_est)
knn_mod = caret::knnreg(dl_mod_form, data = pl_est)
rf_mod  = randomForest(dl_mod_form, data = pl_est)
rp_mod = rpart(dl_mod_form, data = pl_est)
```

- TODO: Note: we are not using `Name`. Why? We are not using `Total`. Why?
- TODO: look what happens with `Total`! You'll see it with `lm()`, you'll be

optimistic with `randomForest()`.

- TODO: What variables are allowed? (With respect to real world problem.)
- TODO: What variables lead to the best predictions?

2.5 Model Evaluation




```
calc_rmse = function(actual, predicted) {
  sqrt(mean( (actual - predicted) ^ 2) )
}
```

```
c(calc_rmse(actual = pl_val$Deadlift, predicted = predict(lm_mod, pl_val)),
  calc_rmse(actual = pl_val$Deadlift, predicted = predict(knn_mod, pl_val)),
  calc_rmse(actual = pl_val$Deadlift, predicted = predict(rp_mod, pl_val)),
  calc_rmse(actual = pl_val$Deadlift, predicted = predict(rf_mod, pl_val)))
```

```
## [1] 18.26654 19.19625 21.68142 19.23643
```

```
reg_preds = map(list(lm_mod, knn_mod, rp_mod, rf_mod), predict, pl_val)
map_dbl(reg_preds, calc_rmse, actual = pl_val$Deadlift)
```

```
## [1] 18.26654 19.19625 21.68142 19.23643
```

- TODO: **Never** supply `data = df` to `predict()`. You have been warned.

```
knitr::include_graphics("img/sim-city.jpg")
```


```
calc_mae = function(actual, predicted) {
  mean(abs(actual - predicted))
}
```

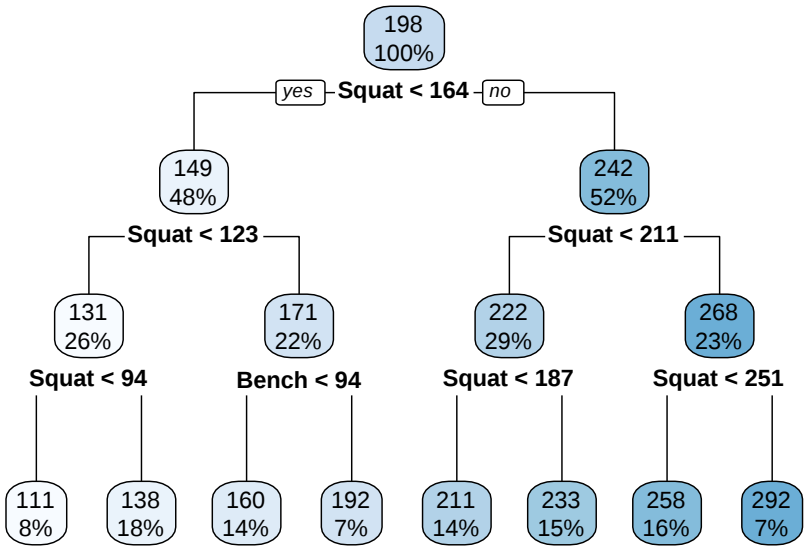
```
map_dbl(reg_preds, calc_mae,  actual = pl_val$Deadlift)
```

```
## [1] 14.38953 14.99748 17.14823 15.28626
```

```
reg_results = tibble(
  Model = c("Linear", "KNN", "Tree", "Forest"),
  RMSE = map_dbl(reg_preds, calc_rmse, actual = pl_val$Deadlift),
  MAE = map_dbl(reg_preds, calc_mae,  actual = pl_val$Deadlift))
```

Model	RMSE	MAE
Linear	18.26654	14.38953
KNN	19.19625	14.99748
Tree	21.68142	17.14823
Forest	19.23643	15.28626

2.6 Discussion




```
lm_mod_final = lm(dl_mod_form, data = pl_trn)
```

```
calc_rmse(actual = pl_tst$Deadlift,
          predicted = predict(lm_mod_final, pl_tst))
```

```
## [1] 22.29668
```

- TODO: Is this a good model?
- TODO: Is this model useful?

```
william_biscarri = tibble(
  Name = "William Biscarri",
  Age = 28,
  Sex = "M",
  Bodyweight = 83,
  Squat = 130,
  Bench = 90
)
```

```
predict(lm_mod_final, william_biscarri)
```

```
##       1
## 175.495
```

Chapter 3

Classification: Handwriting

```
library(readr)
library(tibble)
library(dplyr)
library(purrr)
library(ggplot2)
library(ggridges)
library(lubridate)
library(randomForest)
library(rpart)
library(rpart.plot)
library(cluster)
library(caret)
library(factoextra)
library(rsample)
library(janitor)
library(rvest)
library(dendextend)
library(knitr)
library(kableExtra)
library(ggthemes)
```

- TODO: Show package messaging? check conflicts!
- TODO: Should this be split into three analyses with different packages?

3.1 Background

- TODO: https://en.wikipedia.org/wiki/MNIST_database

- TODO: http://yann.lecun.com/exdb/mnist/

3.2 Data

- TODO: How is this data pre-processed?
- TODO: https://gist.github.com/daviddalpiaz/ae62ae5ccd0bada4b9acd6dbc9008706
- TODO: https://github.com/itsrainingdata/mnistR
- TODO: https://pjreddie.com/projects/mnist-in-csv/
- TODO: http://varianceexplained.org/r/digit-eda/

```
mnist_trn = read_csv(file = "data/mnist_train_subest.csv")
mnist_tst = read_csv(file = "data/mnist_test.csv")
```

```
mnist_trn_y = as.factor(mnist_trn$X1)
mnist_tst_y = as.factor(mnist_tst$X1)

mnist_trn_x = mnist_trn[, -1]
mnist_tst_x = mnist_tst[, -1]
```

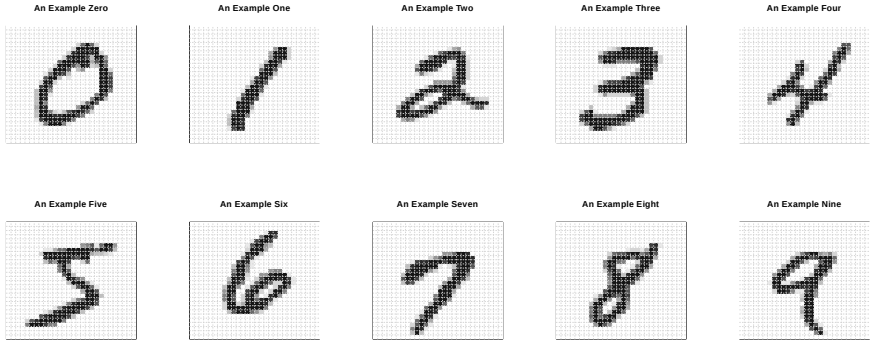
- TODO: If we were going to tune a model, we would need a validation split as well. We're going to be lazy and just fit a single random forest.
- TODO: This is an agreed upon split.

3.3 EDA

```
pixel_positions = expand.grid(j = sprintf("%02.0f", 1:28),
                              i = sprintf("%02.0f", 1:28))
pixel_names = paste("pixel", pixel_positions$i, pixel_positions$j, sep = "-")
```

```
colnames(mnist_trn_x) = pixel_names
colnames(mnist_tst_x) = pixel_names
```

```
show_digit = function(arr784, col = gray(12:1 / 12), ...) {
  image(matrix(as.matrix(arr784), nrow = 28)[, 28:1],
        col = col, xaxt = "n", yaxt = "n", ...)
  grid(nx = 28, ny = 28)
}
```



3.4 Modeling

```
set.seed(42)
mnist_rf = randomForest(x = mnist_trn_x, y = mnist_trn_y, ntree = 100)
```

3.5 Model Evaluation

```
mnist_tst_pred = predict(mnist_rf, mnist_tst_x)
mean(mnist_tst_pred == mnist_tst_y)
```

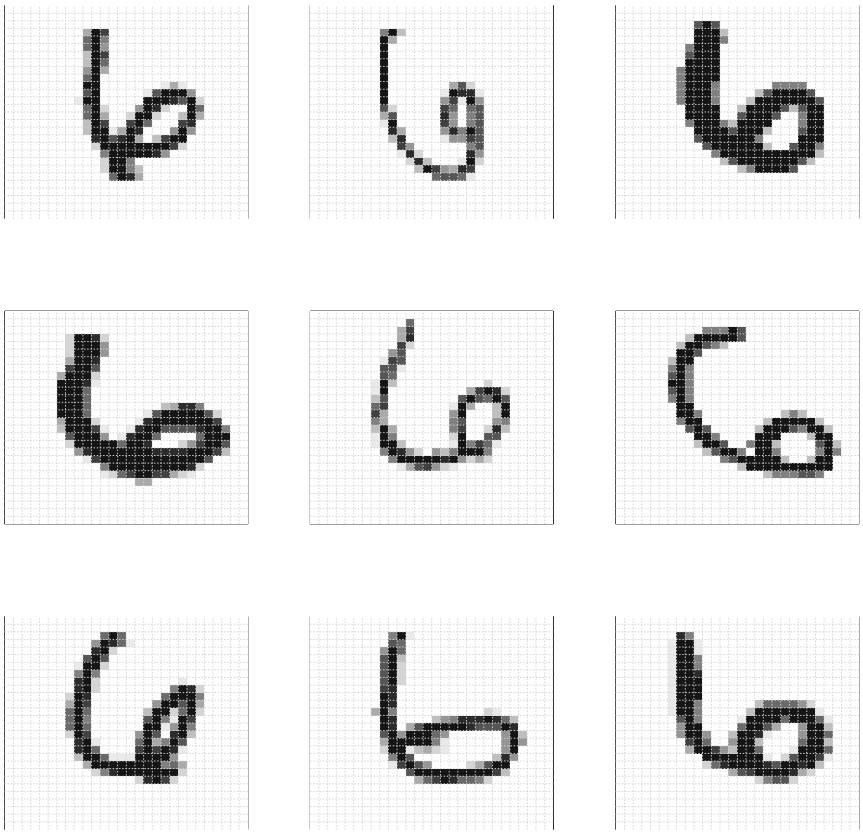
```
## [1] 0.8839
```

```
table(predicted = mnist_tst_pred, actual = mnist_tst_y)
```

```
##           actual
## predicted    0    1    2    3    4    5    6    7    8    9
##         0  959    0   14    6    1   15   22    1   10   10
##         1    0 1112    5    5    1   16    5    9    5    6
##         2    1    2  928   31    3    5   19   24   17    8
##         3    0    2   11  820    1   24    0    1   13   13
##         4    4    0   13    1  839   21   39   11   18   40
##         5    3    1    1   88    3  720   18    1   25    9
##         6    7    2   15    3   25   15  848    0   18    2
##         7    2    1   29   24    1   14    2  928   15   30
##         8    4   14   13   22    5   19    5    4  797    3
##         9    0    1    3   10  103   43    0   49   56  888
```

3.6 Discussion

```
par(mfrow = c(3, 3))
plot_mistake(actual = 6, predicted = 4)
```



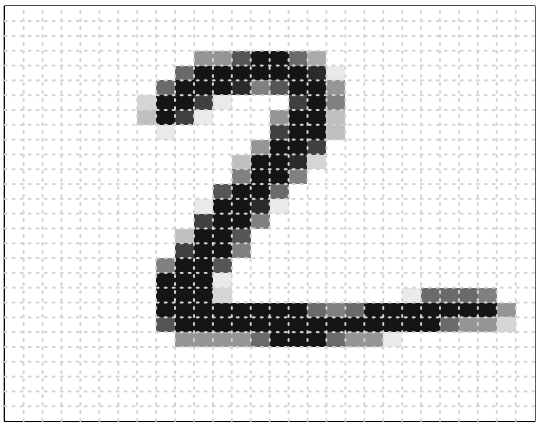
```
mnist_obs_to_check = 2
predict(mnist_rf, mnist_tst_x[mnist_obs_to_check, ], type = "prob")[1, ]
```

```
##    0    1    2    3    4    5    6    7    8    9
## 0.09 0.03 0.25 0.14 0.02 0.14 0.25 0.01 0.05 0.02
```

```
mnist_tst_y[mnist_obs_to_check]
```

```
## [1] 2
## Levels: 0 1 2 3 4 5 6 7 8 9
```

```
show_digit(mnist_tst_x[mnist_obs_to_check, ])
```



Chapter 4

Clustering: Basketball Players

```
library(readr)
library(tibble)
library(dplyr)
library(purrr)
library(ggplot2)
library(ggridges)
library(lubridate)
library(randomForest)
library(rpart)
library(rpart.plot)
library(cluster)
library(caret)
library(factoextra)
library(rsample)
library(janitor)
library(rvest)
library(dendextend)
library(knitr)
library(kableExtra)
library(ggthemes)
```

- TODO: Show package messaging? check conflicts!
- TODO: Should this be split into three analyses with different packages?

4.1 Background

- https://www.youtube.com/watch?v=cuLprHh_BRg
- https://www.youtube.com/watch?v=1FBwSO_1Mb8
- https://www.basketball-reference.com/leagues/NBA_2019.html
- inspiration here, and others: http://blog.schochastics.net/post/analyzing-nba-player-data-ii-clustering/

4.2 Data

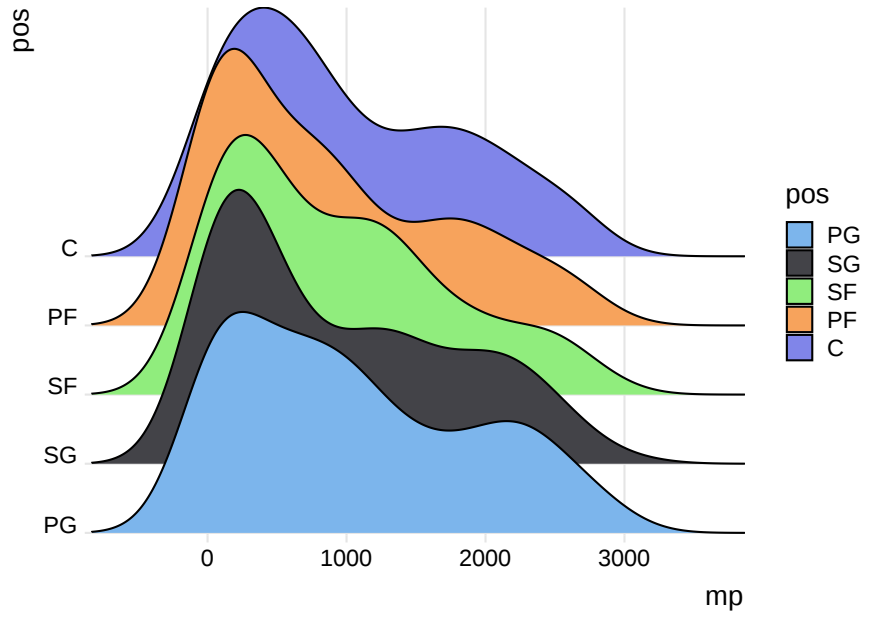
- https://www.basketball-reference.com/leagues/NBA_2019_totals.html
- https://www.basketball-reference.com/leagues/NBA_2019_per_minute.html
- https://www.basketball-reference.com/leagues/NBA_2019_per_poss.html
- https://www.basketball-reference.com/leagues/NBA_2019_advanced.html

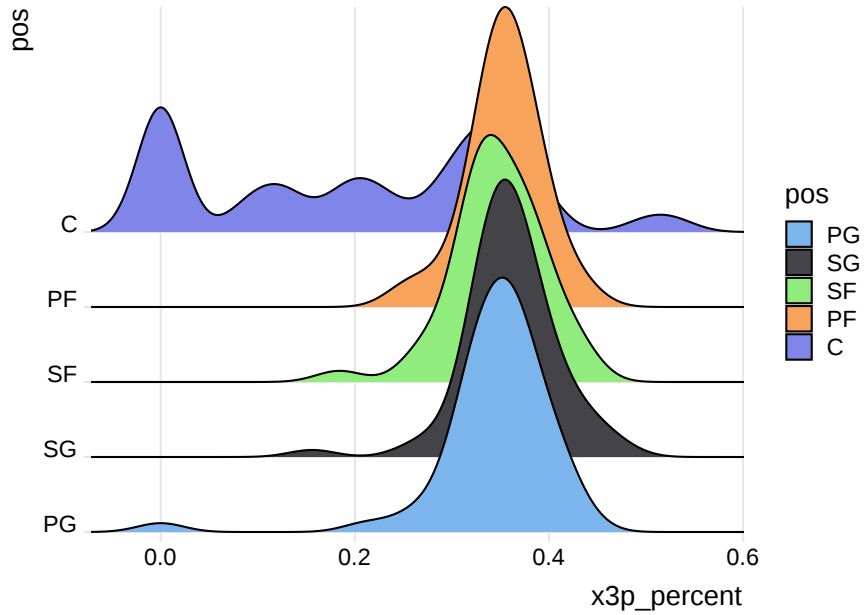
```
nba = scrape_nba_season_player_stats()
nba$pos = factor(nba$pos, levels = c("PG", "SG", "SF", "PF", "C"))
```

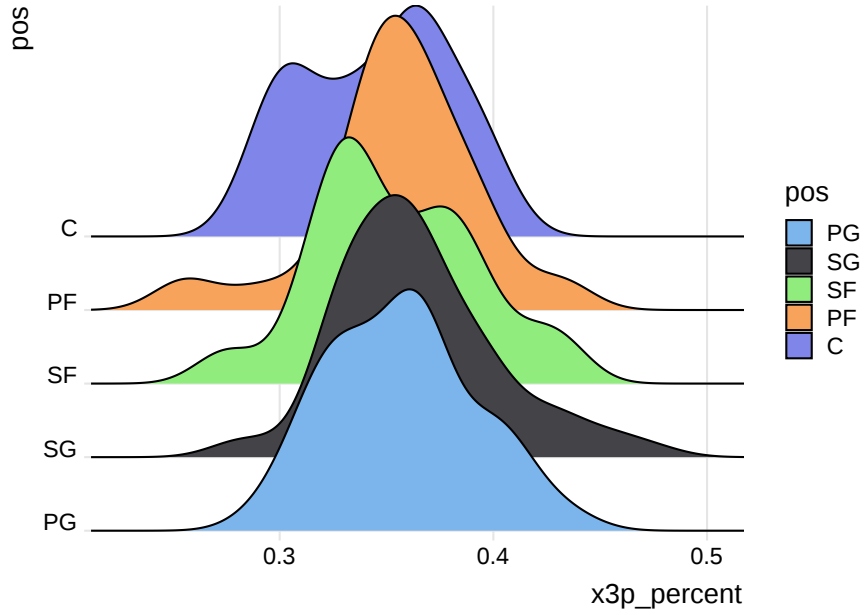
```
## # A tibble: 100 x 93
##    player_team pos     age tm        g    gs    mp    fg   fga fg_percent   x3p
##    <chr>       <fct> <dbl> <chr> <dbl> <dbl> <dbl> <dbl> <dbl>      <dbl> <dbl>
##  1 Álex Abrin~ SG       25 OKC      31     2   588    56   157      0.357    41
##  2 Quincy Acy~ PF       28 PHO      10     0   123     4    18      0.222     2
##  3 Jaylen Ada~ PG       22 ATL      34     1   428    38   110      0.345    25
##  4 Steven Ada~ C        25 OKC      80    80  2669   481   809      0.595     0
##  5 Bam Adebay~ C        21 MIA      82    28  1913   280   486      0.576     3
##  6 Deng Adel ~ SF       21 CLE      19     3   194    11    36      0.306     6
##  7 DeVaughn A~ SG       25 DEN       7     0    22     3    10      0.3       0
##  8 LaMarcus A~ C        33 SAS      81    81  2687   684  1319      0.519    10
##  9 Rawle Alki~ SG       21 CHI      10     1   120    13    39      0.333     3
## 10 Grayson Al~ SG       23 UTA      38     2   416    67   178      0.376    32
## # ... with 90 more rows, and 82 more variables: x3pa <dbl>, x3p_percent <dbl>,
## #   x2p <dbl>, x2pa <dbl>, x2p_percent <dbl>, e_fg_percent <dbl>, ft <dbl>,
## #   fta <dbl>, ft_percent <dbl>, orb <dbl>, drb <dbl>, trb <dbl>, ast <dbl>,
## #   stl <dbl>, blk <dbl>, tov <dbl>, pf <dbl>, pts <dbl>, fg_pm <dbl>,
## #   fga_pm <dbl>, fg_percent_pm <dbl>, x3p_pm <dbl>, x3pa_pm <dbl>,
## #   x3p_percent_pm <dbl>, x2p_pm <dbl>, x2pa_pm <dbl>, x2p_percent_pm <dbl>,
## #   ft_pm <dbl>, fta_pm <dbl>, ft_percent_pm <dbl>, orb_pm <dbl>, drb_pm <dbl>,
## #   trb_pm <dbl>, ast_pm <dbl>, stl_pm <dbl>, blk_pm <dbl>, tov_pm <dbl>,
## #   pf_pm <dbl>, pts_pm <dbl>, fg_pp <dbl>, fga_pp <dbl>, fg_percent_pp <dbl>,
## #   x3p_pp <dbl>, x3pa_pp <dbl>, x3p_percent_pp <dbl>, x2p_pp <dbl>,
```

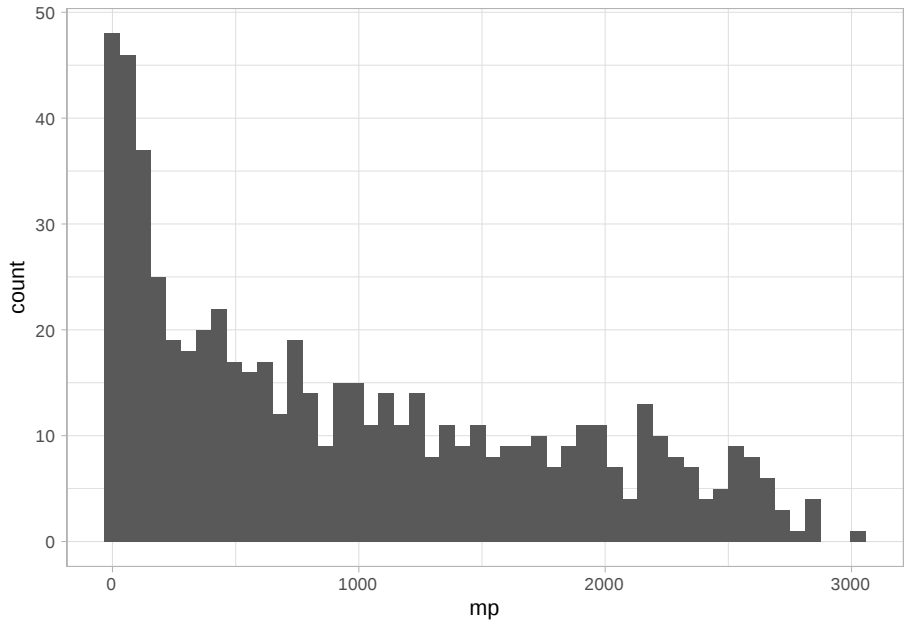
```
## #   x2pa_pp <dbl>, x2p_percent_pp <dbl>, ft_pp <dbl>, fta_pp <dbl>,
## #   ft_percent_pp <dbl>, orb_pp <dbl>, drb_pp <dbl>, trb_pp <dbl>,
## #   ast_pp <dbl>, stl_pp <dbl>, blk_pp <dbl>, tov_pp <dbl>, pf_pp <dbl>,
## #   pts_pp <dbl>, o_rtg_pp <dbl>, d_rtg_pp <dbl>, per <dbl>, ts_percent <dbl>,
## #   x3p_ar <dbl>, f_tr <dbl>, orb_percent <dbl>, drb_percent <dbl>,
## #   trb_percent <dbl>, ast_percent <dbl>, stl_percent <dbl>, blk_percent <dbl>,
## #   tov_percent <dbl>, usg_percent <dbl>, ows <dbl>, dws <dbl>, ws <dbl>,
## #   ws_48 <dbl>, obpm <dbl>, dbpm <dbl>, bpm <dbl>, vorp <dbl>
```

4.3 EDA




pos
C
PF
SF
SG
PG
0.0
0.2
0.4
0.6
x3p_percent
pos
PG
SG
SF
PF
C


pos
C
PF
SF
SG
PG
0.3
0.4
0.5
x3p_percent
pos
PG
SG
SF
PF
C

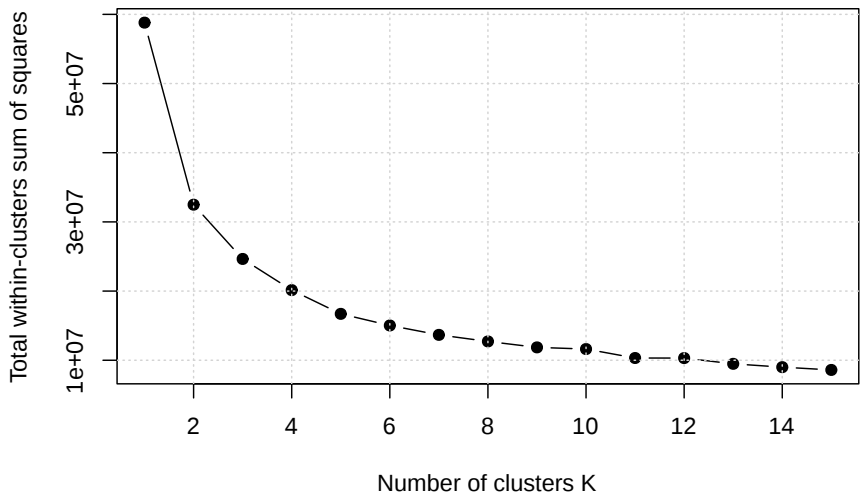



```
nba_for_clustering = nba %>%
  filter(mp > 2000) %>%
  column_to_rownames("player_team") %>%
  select(-pos, -tm)
```

4.4 Modeling

```
set.seed(42)

# function to compute total within-cluster sum of square
wss = function(k, data) {
  kmeans(x = data, centers = k, nstart = 10)$tot.withinss
}

# Compute and plot wss for k = 1 to k = 15
k_values = 1:15

# extract wss for 2-15 clusters
wss_values = map_dbl(k_values, wss, data = nba_for_clustering)

plot(k_values, wss_values,
       type = "b", pch = 19, frame = TRUE,
       xlab = "Number of clusters K",
```

```
       ylab = "Total within-clusters sum of squares")
grid()
```

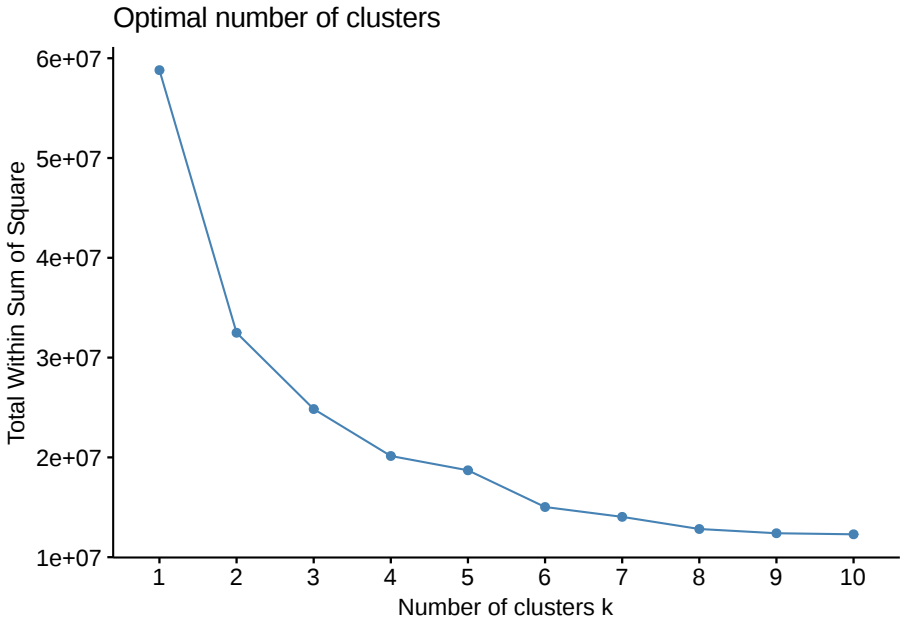


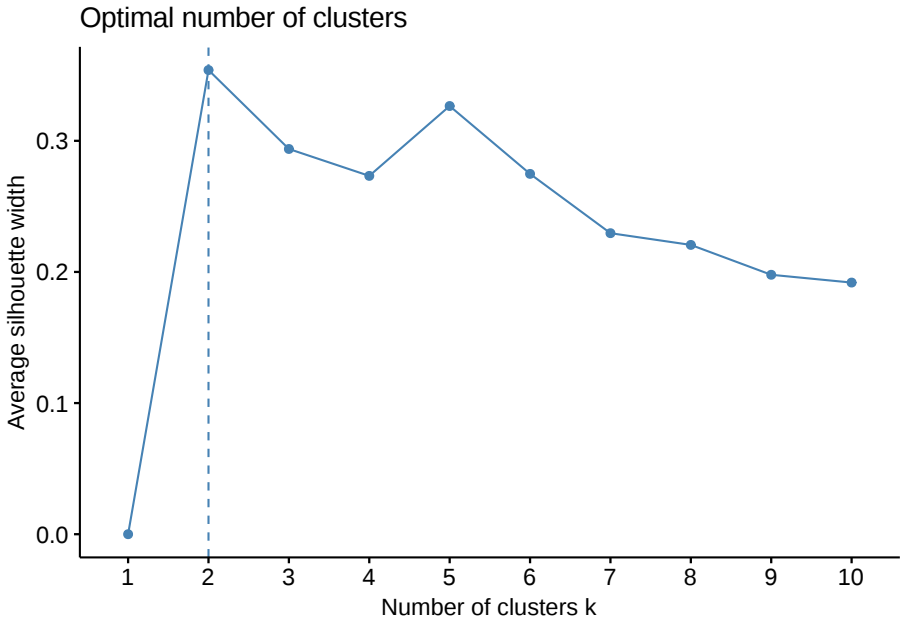
- TODO: K-Means likes clusters of roughly equal size.
- TODO: http://varianceexplained.org/r/kmeans-free-lunch/

```
nba_hc = hclust(dist(nba_for_clustering))
nba_hc_clust = cutree(nba_hc, k = 5)
table(nba_hc_clust)
```

```
## nba_hc_clust
##  1  2  3  4  5
## 38 13 28 11  1
```

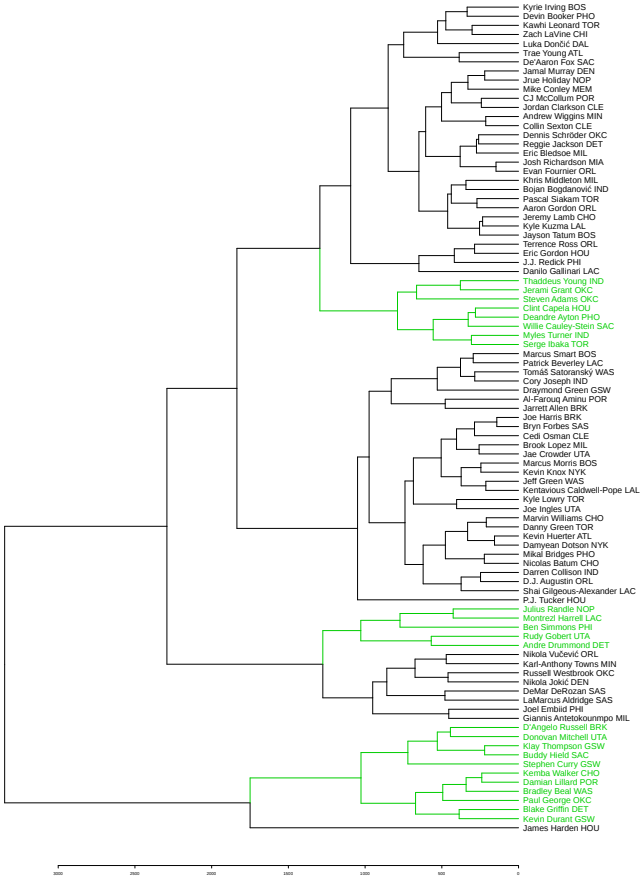
4.5 Model Evaluation






Cluster plot
Dim1 (35.9%)
Dim2 (23.2%)
10
5
0
-5
-10
-15
-10
0
cluster
1
2
3
4
5
Rudy Gobert UTA
Clint Capela HOU
Jarrett Allen BRK
Andre Drummond DET
Deandre Ayton PHO
Willie Cauley-Stein SAC
Draymond Green GSW
P.J. Tucker HOU
Myles Turner IND
Serge Ibaka TOR
Montrezl Harrell LAC
Brook Lopez MIL
Nicolas Batum CHO
Danny Green TOR
Ben Simmons PHI
Pascal Siakam TOR
Marcus Morris BOS
Aaron Gordon ORL
Joe Ingles UTA
Jayson Tatum BOS
LaMarcus Aldridge SAS
Darren Collison IND
Nikola Vučević ORL
Kyle Kuzma LAL
D.J. Augustin ORL
Julius Randle NOP
Eric Bledsoe MIL
Karl-Anthony Towns MIN
Kyle Lowry TOR
Bojan Bogdanović IND
Nikola Jokić DEN
Collin Sexton CLE
Joel Embiid PHI
DeMar DeRozan SAS
CJ McCollum POR
Klay Thompson GSW
Buddy Hield SAC
Zach LaVine CHI
Kevin Durant GSW
Blake Griffin DET
Mike Conley MEM
Donovan Mitchell UTA
Trae Young ATL
Paul George OKC
D'Angelo Russell BRK
Stephen Curry GSW
James Harden HOU

4.6 Discussion



Part II

Some Machine Learning Foundations

Chapter 5

Introduction

- TODO: This section could ask be called a “detour”

Chapter 6

Probability

- TODO: Note! This is copy-pasted from R4SL.

We give a very brief review of some necessary probability concepts. As the treatment is less than complete, a list of references is given at the end of the chapter. For example, we ignore the usual recap of basic set theory and omit proofs and examples.

6.1 Probability Models

When discussing probability models, we speak of random **experiments** that produce one of a number of possible **outcomes**.

A **probability model** that describes the uncertainty of an experiment consists of two elements:

- The **sample space**, often denoted as Ω, which is a set that contains all possible outcomes.
- A **probability function** that assigns to an event A a nonnegative number, $P[A]$, that represents how likely it is that event A occurs as a result of the experiment.

We call $P[A]$ the **probability** of event A. An **event** A could be any subset of the sample space, not necessarily a single possible outcome. The probability law must follow a number of rules, which are the result of a set of axioms that we introduce now.

6.2 Probability Axioms

Given a sample space Ω for a particular experiment, the **probability function** associated with the experiment must satisfy the following axioms.

1. *Nonnegativity*: $P[A] \geq 0$ for any event $A \subset \Omega$.
2. *Normalization*: $P[\Omega] = 1$. That is, the probability of the entire space is 1.
3. *Additivity*: For mutually exclusive events $E_1, E_2, \ldots$

$$P\left[\bigcup_{i=1}^{\infty} E_i\right] = \sum_{i=1}^{\infty} P[E_i]$$

Using these axioms, many additional probability rules can easily be derived.

6.3 Probability Rules

Given an event A, and its complement, A^c, that is, the outcomes in Ω which are not in A, we have the **complement rule**:

$$P[A^c] = 1 - P[A]$$

In general, for two events A and B, we have the **addition rule**:

$$P[A \cup B] = P[A] + P[B] - P[A \cap B]$$

If A and B are also *disjoint*, then we have:

$$P[A \cup B] = P[A] + P[B]$$

If we have n mutually exclusive events, $E_1, E_2, \ldots E_n$, then we have:

$$P\left[\bigcup_{i=1}^{n} E_i\right] = \sum_{i=1}^{n} P[E_i]$$

Often, we would like to understand the probability of an event A, given some information about the outcome of event B. In that case, we have the **conditional probability rule** provided $P[B] > 0$.

$$P[A \mid B] = \frac{P[A \cap B]}{P[B]}$$

Rearranging the conditional probability rule, we obtain the **multiplication rule**:

$$P[A \cap B] = P[B] \cdot P[A \mid B]\cdot$$

For a number of events $E_1, E_2, \ldots E_n$, the multiplication rule can be expanded into the **chain rule**:

$$P\left[\bigcap_{i=1}^{n} E_i\right] = P[E_1] \cdot P[E_2 \mid E_1] \cdot P[E_3 \mid E_1 \cap E_2] \cdots P\left[E_n \mid \bigcap_{i=1}^{n-1} E_i\right]$$

Define a **partition** of a sample space Ω to be a set of disjoint events $A_1, A_2, \ldots, A_n$ whose union is the sample space Ω. That is

$$A_i \cap A_j = \emptyset$$

for all $i \neq j$, and

$$\bigcup_{i=1}^{n} A_i = \Omega.$$

Now, let $A_1, A_2, \ldots, A_n$ form a partition of the sample space where $P[A_i] > 0$ for all i. Then for any event B with $P[B] > 0$ we have **Bayes' Rule**:

$$P[A_i|B] = \frac{P[A_i]P[B|A_i]}{P[B]} = \frac{P[A_i]P[B|A_i]}{\sum_{i=1}^{n} P[A_i]P[B|A_i]}$$

The denominator of the latter equality is often called the **law of total probability**:

$$P[B] = \sum_{i=1}^{n} P[A_i]P[B|A_i]$$

Two events A and B are said to be **independent** if they satisfy

$$P[A \cap B] = P[A] \cdot P[B]$$

This becomes the new multiplication rule for independent events.

A collection of events $E_1, E_2, \ldots E_n$ is said to be independent if

$$P\left[\bigcap_{i\in S} E_i\right] = \prod_{i\in S} P[E_i]$$

for every subset S of $\{1, 2, \ldots n\}$.

If this is the case, then the chain rule is greatly simplified to:

$$P\left[\bigcap_{i=1}^{n} E_i\right] = \prod_{i=1}^{n} P[E_i]$$

6.4 Random Variables

A **random variable** is simply a *function* which maps outcomes in the sample space to real numbers.

6.4.1 Distributions

We often talk about the **distribution** of a random variable, which can be thought of as:

$$\text{distribution} = \text{list of possible } \textbf{values} + \text{associated } \textbf{probabilities}$$

This is not a strict mathematical definition, but is useful for conveying the idea.

If the possible values of a random variables are *discrete*, it is called a *discrete random variable*. If the possible values of a random variables are *continuous*, it is called a *continuous random variable*.

6.4.2 Discrete Random Variables

The distribution of a discrete random variable X is most often specified by a list of possible values and a probability **mass** function, $p(x)$. The mass function directly gives probabilities, that is,

$$p(x) = p_X(x) = P[X = x].$$

Note we almost always drop the subscript from the more correct $p_X(x)$ and simply refer to $p(x)$. The relevant random variable is discerned from context

The most common example of a discrete random variable is a **binomial** random variable. The mass function of a binomial random variable X, is given by

$$p(x|n,p) = \binom{n}{x} p^x (1-p)^{n-x}, \quad x = 0, 1, \ldots, n, \ n \in \mathbb{N}, \ 0 < p < 1.$$

This line conveys a large amount of information.

- The function $p(x|n,p)$ is the mass function. It is a function of x, the possible values of the random variable X. It is conditional on the **parameters** n and p. Different values of these parameters specify different binomial distributions.
- $x = 0, 1, \ldots, n$ indicates the **sample space**, that is, the possible values of the random variable.
- $n \in \mathbb{N}$ and $0 < p < 1$ specify the **parameter spaces**. These are the possible values of the parameters that give a valid binomial distribution.

Often all of this information is simply encoded by writing

$$X \sim \text{bin}(n,p).$$

6.4.3 Continuous Random Variables

The distribution of a continuous random variable X is most often specified by a set of possible values and a probability **density** function, $f(x)$. (A cumulative density or moment generating function would also suffice.)

The probability of the event $a < X < b$ is calculated as

$$P[a < X < b] = \int_a^b f(x)dx.$$

Note that densities are **not** probabilities.

The most common example of a continuous random variable is a **normal** random variable. The density of a normal random variable X, is given by

$$f(x|\mu,\sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} \cdot \exp\left[\frac{-1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right], \quad -\infty < x < \infty, \ -\infty < \mu < \infty, \ \sigma > 0.$$

- The function $f(x|\mu,\sigma^2)$ is the density function. It is a function of x, the possible values of the random variable X. It is conditional on the **paramters** μ and σ^2. Different values of these parameters specify different normal distributions.
- $-\infty < x < \infty$ indicates the sample space. In this case, the random variable may take any value on the real line.

- $-\infty < \mu < \infty$ and $\sigma > 0$ specify the parameter space. These are the possible values of the parameters that give a valid normal distribution.

Often all of this information is simply encoded by writing

$$X \sim N(\mu, \sigma^2)$$

6.4.4 Several Random Variables

Consider two random variables X and Y. We say they are independent if

$$f(x, y) = f(x) \cdot f(y)$$

for all x and y. Here $f(x, y)$ is the **joint** density (mass) function of X and Y. We call $f(x)$ the **marginal** density (mass) function of X. Then $f(y)$ the marginal density (mass) function of Y. The joint density (mass) function $f(x, y)$ together with the possible (x, y) values specify the joint distribution of X and Y.

Similar notions exist for more than two variables.

6.5 Expectations

For discrete random variables, we define the **expectation** of the function of a random variable X as follows.

$$\mathbb{E}[g(X)] \triangleq \sum_x g(x)p(x)$$

For continuous random variables we have a similar definition.

$$\mathbb{E}[g(X)] \triangleq \int g(x)f(x)dx$$

For specific functions g, expectations are given names.

The **mean** of a random variable X is given by

$$\mu_X = \text{mean}[X] \triangleq \mathbb{E}[X].$$

So for a discrete random variable, we would have

$$\text{mean}[X] = \sum_x x \cdot p(x)$$

For a continuous random variable we would simply replace the sum by an integral.

The **variance** of a random variable X is given by

$$\sigma_X^2 = \text{var}[X] \triangleq \mathbb{E}[(X - \mathbb{E}[X])^2] = \mathbb{E}[X^2] - (\mathbb{E}[X])^2.$$

The **standard deviation** of a random variable X is given by

$$\sigma_X = \text{sd}[X] \triangleq \sqrt{\sigma_X^2} = \sqrt{\text{var}[X]}.$$

The **covariance** of random variables X and Y is given by

$$\text{cov}[X, Y] \triangleq \mathbb{E}[(X - \mathbb{E}[X])(Y - \mathbb{E}[Y])] = \mathbb{E}[XY] - \mathbb{E}[X] \cdot \mathbb{E}[Y].$$

6.6 Likelihood

Consider n iid random variables $X_1, X_2, \ldots X_n$. We can then write their **likelihood** as

$$\mathcal{L}(\theta \mid x_1, x_2, \ldots x_n) = \prod_{i=i}^{n} f(x_i; \theta)$$

where $f(x_i; \theta)$ is the density (or mass) function of random variable X_i evaluated at x_i with parameter θ.

Whereas a probability is a function of a possible observed value given a particular parameter value, a likelihood is the opposite. It is a function of a possible parameter value given observed data.

Maximumizing likelihood is a common techinque for fitting a model to data.

6.7 Videos

The YouTube channel mathematicalmonk has a great Probability Primer playlist containing lectures on many fundamental probability concepts. Some of the more important concepts are covered in the following videos:

- Conditional Probability
- Independence
- More Independence
- Bayes Rule

6.8 References

Any of the following are either dedicated to, or contain a good coverage of the details of the topics above.

- Probability Texts
  - Introduction to Probability by Dimitri P. Bertsekas and John N. Tsitsiklis
  - A First Course in Probability by Sheldon Ross
- Machine Learning Texts with Probability Focus
  - Probability for Statistics and Machine Learning by Anirban DasGupta
  - Machine Learning: A Probabilistic Perspective by Kevin P. Murphy
- Statistics Texts with Introduction to Probability
  - Probability and Statistical Inference by Robert V. Hogg, Elliot Tanis, and Dale Zimmerman
  - Introduction to Mathematical Statistics by Robert V. Hogg, Joseph McKean, and Allen T. Craig

Chapter 7

Estimation

- TODO: Where we are going, estimating conditional means and distributions.
- TODO: estimation = learning. "learning from data." what are we learning about? often parameters.
- TODO: http://stat400.org
- TODO: http://stat420.org

7.1 Probability

- TODO: See Appendix A
- TODO: In R, `d*()`, `p*()`, `q*()`, `r*()`

7.2 Statistics

- TODO: parameters are a function of the population distribution
- TODO: statistics are a function of data.
- TODO: parameters:population::statistics::data
- TODO: statistic vs value of a statistic

7.3 Estimators

- TODO: estimator vs estimate
- TODO: Why such a foucs on the mean, E[X]? Because E[(X - a)^2] is minimized by E[X]
  - https://www.benkuhn.net/squared

– https://news.ycombinator.com/item?id=9556459

7.3.1 Properties

7.3.1.1 Bias

$$\text{bias}\left[\hat{\theta}\right] \triangleq \mathbb{E}\left[\hat{\theta}\right] - \theta$$

7.3.1.2 Variance

$$\text{var}\left[\hat{\theta}\right] \triangleq \mathbb{E}\left[\left(\hat{\theta} - \mathbb{E}\left[\hat{\theta}\right]\right)^2\right]$$

7.3.1.3 Mean Squared Error

$$\text{MSE}\left[\hat{\theta}\right] \triangleq \mathbb{E}\left[\left(\hat{\theta} - \theta\right)^2\right] = \text{var}\left[\hat{\theta}\right] + \left(\text{Bias}\left[\hat{\theta}\right]\right)^2$$

7.3.1.4 Consistency

An estimator $\hat{\theta}_n$ is said to be a **consistent estimator** of θ if, for any positive ϵ,

$$\lim_{n\to\infty} P\left(\left|\hat{\theta}_n - \theta\right| \leq \epsilon\right) = 1$$

or, equivalently,

$$\lim_{n\to\infty} P\left(\left|\hat{\theta}_n - \theta\right| > \epsilon\right) = 0$$

We say that $\hat{\theta}_n$ **converges in probability** to θ and we write $\hat{\theta}_n \xrightarrow{P} \theta$.

7.3.2 Methods

- TODO: MLE

Given a random sample $X_1, X_2, \ldots, X_n$ from a population with parameter θ and density or mass $f(x \mid \theta)$, we have:

The Likelihood, $L(\theta)$,

$$L(\theta) = f(x_1, x_2, \ldots, x_n) = \prod_{i=1}^{n} f(x_i \mid \theta)$$

The **Maximum Likelihood Estimator**, $\hat{\theta}$

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}}\ L(\theta) = \underset{\theta}{\operatorname{argmax}}\ \log L(\theta)$$

- TODO: Invariance Principle

If $\hat{\theta}$ is the MLE of θ and the function $h(\theta)$ is continuous, then $h(\hat{\theta})$ is the MLE of $h(\theta)$.

- TODO: MOM
- TODO: https://daviddalpiaz.github.io/stat3202-sp19/notes/fitting.html
- TODO: ECDF: https://en.wikipedia.org/wiki/Empirical_distribution_function

Part III

A Tour of Machine Learning

Chapter 8

Introduction

- TODO: **Goal:** Train models that *generalize* well, that is, perform well on *unseen* data.
- TODO: Introduce data splitting:
  - Test-Train Split (`_tst` and `_trn`)
  - Estimation-Validation Split (`_est` and `_val`)
    - Where are these "weird" terms coming from?
    - Why not cross-validation yet?
- TODO: http://varianceexplained.org/r/ds-ml-ai/s
- TODO: define most of the terms that will be seen here
  - at least those that apply to both regression and classification
- TODO: This section is the heart of STAT 432.
- TODO: Note simplifications that we will use in this section.
  - No cross-validation
  - No data pre-processing
  - Minimial care for categorical variables
    - Either don't use them, or let the methods take care of them.

Chapter 9

Regression

BLUF: Use **regression**, which is one of the two **supervised learning** tasks (the other being **classification**) to make predictions of new observations of **numeric *response* variables.** Start by randomly splitting the data (which includes both the response and the **features**) into a **test set** and a **training set**. Do not use the test data for anything other than supplying a final assesment of how well a chosen model performs at the prediction task. That is, never use the test data to make *any* modeling decisions. Use the training data however you please, but it is recommended to further split this data into an **estimation set** and a **validation set**. The estimation set should be used to **train** models for evaluation. For example, use the estimation data to learn the **model parameters** of a **parametric model**. Do not use data used in training of models (the estimation data) when evaluating models as doing so will mask **overfitting** of **complex** (flexible) models. Use the `lm()` function to train **linear models**. Use the `knnreg()` function from the `caret` pacakge to train **k-nearest neighbors models**. Use the `rpart()` function from the `rpart` package to train **decision tree models**. Use the validation set to evaluate models that have been trained using the estimation data. For example, use the validation data to select the value of **tuning parameters** that are often used in **non-parametric models**. Use numeric metrics such as **root-mean-square error (RMSE)** or graphical summaries such as **actual versus predicted plots**. Although it ignores some practical and statistical considerations (which will be discussed later), the model that acheives the lowest RMSE on the validation data will be deemed the "best" model. After finding this model, refit the model to the entire training dataset. Report the RMSE of this model on the test data as a final quantification of performance.

- TODO: add ISL readings
- TODO: <www.stat420.org>
- TODO: add "why least squares?" readings

```
library("tidyverse")
library("caret")
library("rpart")
library("knitr")
library("kableExtra")
```

9.1 Setup

$$Y = f(\boldsymbol{X}) + \epsilon$$

- TODO: signal $f(X)$
- TODO: noise ϵ
- TODO: goal: learn the signal, not the noise
- TODO: random variables versus potential realized values

$$\boldsymbol{X} = (X_1, X_2, \ldots, X_p)$$

$$\boldsymbol{x} = (x_1, x_2, \ldots, x_p)$$

$$\mathbb{E}\left[\left(Y - f(\boldsymbol{X})^2\right)\right]$$

- TODO: define regression function
  - above is minimzed when f(x) = mu(x)

$$\mu(\boldsymbol{x}) = \mathbb{E}[Y \mid \boldsymbol{X} = \boldsymbol{x}]$$

- TODO: want to learn these "things" which are regression functions

```
line_reg_fun = function(x) {
  x
}
```

$$\mu_l(x) = x$$

```
quad_reg_fun = function(x) {
  x ^ 2
}
```

$$\mu_q(x) = x^2$$

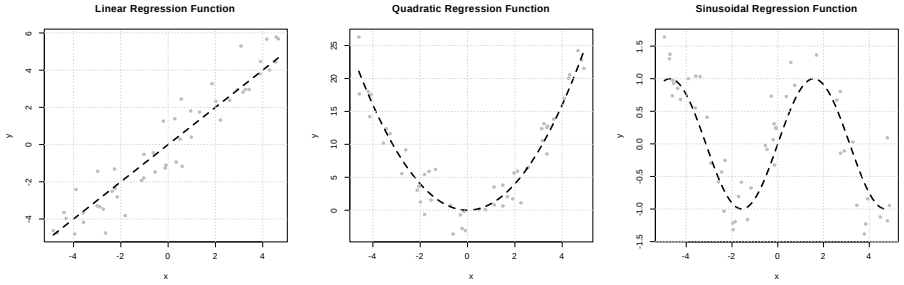
```
sine_reg_fun = function(x) {
  sin(x)
}
```

$$\mu_s(x) = \sin(x)$$

```
gen_sim_data = function(f, sample_size = 50, sd = 1) {
  x = runif(n = sample_size, min = -5, max = 5)
  y = rnorm(n = sample_size, mean = f(x), sd = sd)
  tibble::tibble(x = x, y = y)
}
```

```
set.seed(5)
line_data = gen_sim_data(f = line_reg_fun, sample_size = 50, sd = 1.0)
quad_data = gen_sim_data(f = quad_reg_fun, sample_size = 50, sd = 2.0)
sine_data = gen_sim_data(f = sine_reg_fun, sample_size = 50, sd = 0.5)
```

```
set.seed(42)
line_data_unseen = gen_sim_data(f = line_reg_fun, sample_size = 100000, sd = 1.0)
quad_data_unseen = gen_sim_data(f = quad_reg_fun, sample_size = 100000, sd = 2.0)
sine_data_unseen = gen_sim_data(f = sine_reg_fun, sample_size = 100000, sd = 0.5)
```



9.2 Modeling

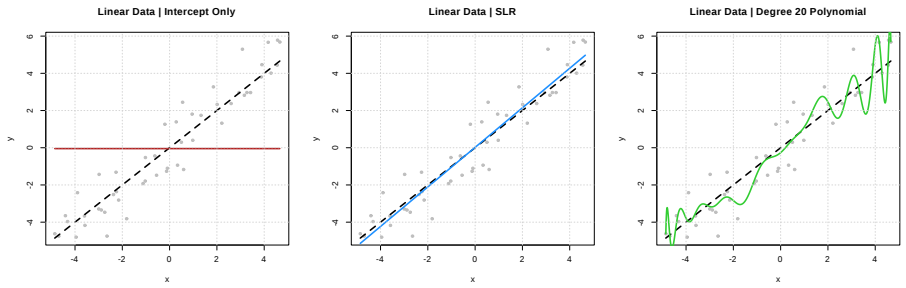
- TODO: for now, only use formula syntax
  - https://rviews.rstudio.com/2017/02/01/the-r-formula-method-the-good-parts/
  - https://rviews.rstudio.com/2017/03/01/the-r-formula-method-the-bad-parts/

9.2.1 Linear Models

- TODO: assume form of mean relationship. linear combination
- TODO: how to go from y = b0 + b1x1 + ... + eps to `lm(y ~ stuff)`

- TODO: least squares, least squares is least squares (difference in assumptions)

```
lm_line_int = lm(y ~ 1,                    data = line_data)
lm_line_slr = lm(y ~ poly(x, degree = 1),  data = line_data)
lm_line_ply = lm(y ~ poly(x, degree = 20), data = line_data)
```



9.2.2 k-Nearest Neighbors

- TODO: `caret::knnreg()`
- TODO: for now, don't worry about scaling, factors, etc.

9.2.2.1 Linear Data

```
knn_line_25 = knnreg(y ~ x, data = line_data, k = 25)
knn_line_05 = knnreg(y ~ x, data = line_data, k = 5)
knn_line_01 = knnreg(y ~ x, data = line_data, k = 1)
```

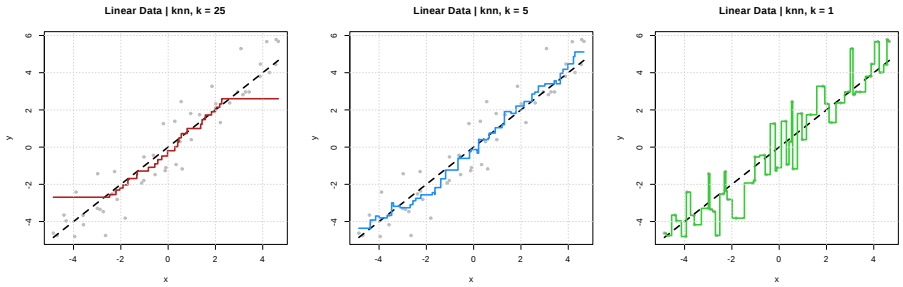
```
calc_dist = function(p1, p2) {
  sqrt(sum((p1 - p2) ^ 2))
}
```

```
line_data %>%
  mutate(dist = purrr::map_dbl(x, calc_dist, p2 = 0)) %>%
  top_n(dist, n = -5) %>%
  pull(y) %>%
  mean() # also consider median
```

```
## [1] -0.1310472
```

```
predict(knn_line_05, data.frame(x = 0))
```

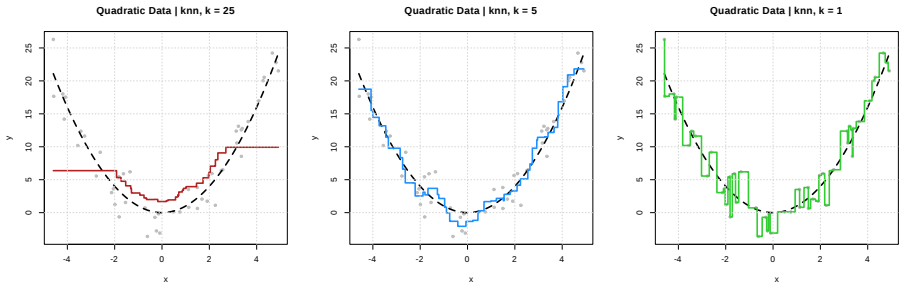
```
## [1] -0.1310472
```



k	Train RMSE	Test RMSE
25	1.406	1.379
5	0.931	1.061
1	0.000	1.409

9.2.2.2 Quadratic Data

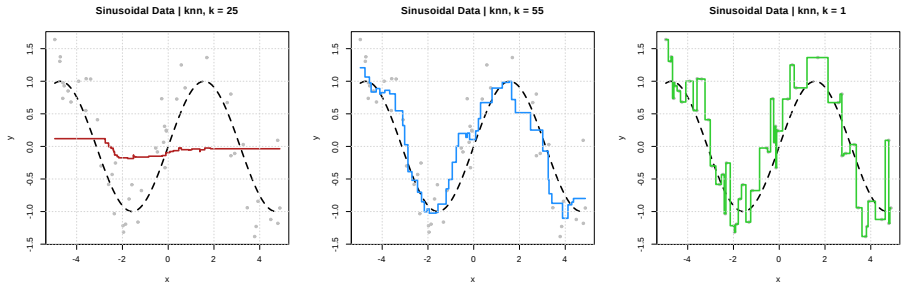
```
knn_quad_25 = knnreg(y ~ x, data = quad_data, k = 25)
knn_quad_05 = knnreg(y ~ x, data = quad_data, k = 5)
knn_quad_01 = knnreg(y ~ x, data = quad_data, k = 1)
```



k	Train RMSE	Test RMSE
25	6.393	6.564
5	2.236	2.509
1	0.000	3.067

9.2.2.3 Sinusoidal Data

```
knn_sine_25 = knnreg(y ~ x, data = sine_data, k = 25)
knn_sine_05 = knnreg(y ~ x, data = sine_data, k = 5)
knn_sine_01 = knnreg(y ~ x, data = sine_data, k = 1)
```



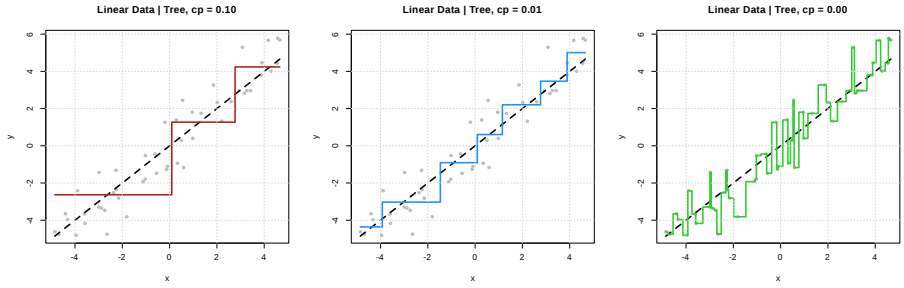
k	Train RMSE	Test RMSE
25	0.814	0.841
5	0.349	0.570
1	0.000	0.647

9.2.3 Decision Trees

- TODO: `rpart::rpart()`
- TODO: https://cran.r-project.org/web/packages/rpart/vignettes/longintro.pdf
- TODO: http://www.milbo.org/doc/prp.pdf
- TODO: maybe notes about pruning and CV

9.2.3.1 Linear Data

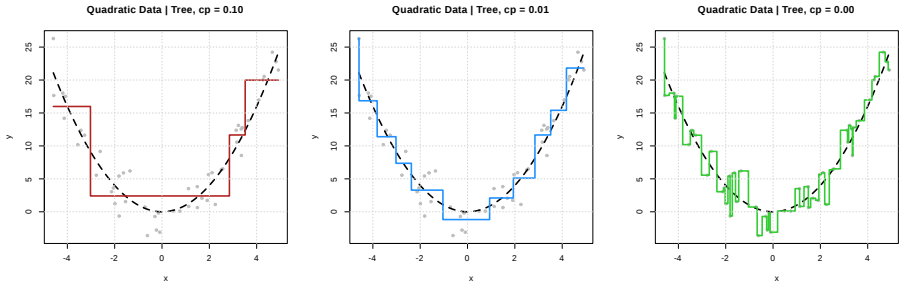
```
tree_line_010 = rpart(y ~ x, data = line_data, cp = 0.10, minsplit = 2)
tree_line_001 = rpart(y ~ x, data = line_data, cp = 0.01, minsplit = 2)
tree_line_000 = rpart(y ~ x, data = line_data, cp = 0.00, minsplit = 2)
```



k	Train RMSE	Test RMSE
25	1.394	1.548
5	0.914	1.144
1	0.000	1.409

9.2.3.2 Quadratic Data

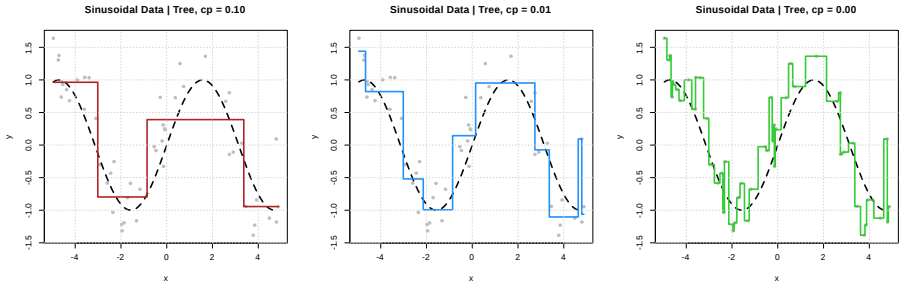
```
tree_quad_010 = rpart(y ~ x, data = quad_data, cp = 0.10, minsplit = 2)
tree_quad_001 = rpart(y ~ x, data = quad_data, cp = 0.01, minsplit = 2)
tree_quad_000 = rpart(y ~ x, data = quad_data, cp = 0.00, minsplit = 2)
```



k	Train RMSE	Test RMSE
25	3.376	3.869
5	1.692	2.621
1	0.000	3.067

9.2.3.3 Sinusoidal Data

```
tree_sine_010 = rpart(y ~ x, data = sine_data, cp = 0.10, minsplit = 2)
tree_sine_001 = rpart(y ~ x, data = sine_data, cp = 0.01, minsplit = 2)
tree_sine_000 = rpart(y ~ x, data = sine_data, cp = 0.00, minsplit = 2)
```



k	Train RMSE	Test RMSE
25	0.414	0.659
5	0.235	0.629
1	0.000	0.647

9.3 Procedure

- TODO: Look at data
- TODO: Pick candiate models
- TODO: Tune / train models
- TODO: Pick "best" model
  - based on validation RMSE (note the issues with this)
- TODO: Use best model / report test metrics

9.4 Data Splitting

- TODO: want to generalize to unseen data
- TODO: for now, all variables should either be numeric, or factor
- TODO: Training (Train) Data
- TODO: Testing (Test) Data
- TODO: Estimation Data
- TODO: Validation Data
  - https://en.wikipedia.org/wiki/Infinite_monkey_theorem

$$\mathcal{D} = \{(x_i, y_i) \in \mathbb{R}^p \times \mathbb{R}, \ i = 1, 2, \ldots n\}$$

$$\mathcal{D} = \mathcal{D}_{\texttt{trn}} \cup \mathcal{D}_{\texttt{tst}}$$

$$\mathcal{D}_{\texttt{trn}} = \mathcal{D}_{\texttt{est}} \cup \mathcal{D}_{\texttt{val}}$$

9.5 Metrics

- TODO: RMSE

$$\text{rmse}\left(\hat{f}_{\texttt{set}}, \mathcal{D}_{\texttt{set}}\right) = \sqrt{\frac{1}{n_{\texttt{set}}} \sum_{i \in \texttt{set}} \left(y_i - \hat{f}_{\texttt{set}}(x_i)\right)^2}$$

$$\text{RMSE}_{\texttt{trn}} = \text{rmse}\left(\hat{f}_{\texttt{est}}, \mathcal{D}_{\texttt{est}}\right) = \sqrt{\frac{1}{n_{\texttt{est}}} \sum_{i \in \texttt{est}} \left(y_i - \hat{f}_{\texttt{est}}(x_i)\right)^2}$$

$$\text{RMSE}_{\texttt{val}} = \text{rmse}\left(\hat{f}_{\texttt{est}}, \mathcal{D}_{\texttt{val}}\right) = \sqrt{\frac{1}{n_{\texttt{val}}} \sum_{i \in \texttt{val}} \left(y_i - \hat{f}_{\texttt{est}}(x_i)\right)^2}$$

$$\text{RMSE}_{\texttt{tst}} = \text{rmse}\left(\hat{f}_{\texttt{trn}}, \mathcal{D}_{\texttt{tst}}\right) = \sqrt{\frac{1}{n_{\texttt{tst}}} \sum_{i \in \texttt{tst}} \left(y_i - \hat{f}_{\texttt{trn}}(x_i)\right)^2}$$

- TODO: MAE
- TODO: MAPE
  - https://en.wikipedia.org/wiki/Mean_absolute_percentage_error
  - but probably don't use

```
calc_rmse = function(model, data, response) {
  actual = data[[response]]
  predicted = predict(model, data)
  sqrt(mean((actual - predicted) ^ 2))
}
```

9.6 Model Complexity

- TODO: what determines the complexity of the above models?
  - lm: terms, xforms, interactions
  - knn: k (also terms, xforms, interactions)
  - tree: cp (with rpart, also others that we'll keep mostly hidden) (also terms, xforms, interactions)

9.7 Overfitting

- TODO: too complex
- TODO: usual picture with training and validation error
- TODO: define for the purposes of this course

9.8 Multiple Features

- TODO: more features = more complex
- TODO: how do the three models add additional features?

9.9 Example Analysis

- TODO: Diamonds analysis
- TODO: model.matrix()

9.10 MISC TODOS

- lex fridman with ian: dataset (represent), model, optimize
  - https://www.youtube.com/watch?v=Z6rxFNMGdn0
- want to minimize E[(y - y_hat)^2]
- predict() creates estimate of E[Y|X] with supplied model

$$\mathbb{E}\left[|Y - f(\boldsymbol{X})|\right]$$

$$m(\boldsymbol{x}) = \mathbb{M}[Y \mid \boldsymbol{X} = \boldsymbol{x}]$$

```
# define a data generating process
gen = function() {
  x = runif(100)
  y = 2 * x + rnorm(100)
  tibble(x, y)
}

# generate and check data
df = gen()

# define midpoint calculation
calc_midpoints = function(x) {
  x = sort(x)
  x[-length(x)] + diff(x) / 2
}

# calculate midpoints
mids = with(df, calc_midpoints(x))
```

```
# calculate mse for a proposed split
calc_mse_split = function(df, cut) {

  left  = dplyr::filter(df, x < cut)
  right = dplyr::filter(df, x > cut)

  mse_left  = with(left,  sum((y - mean(y)) ^ 2))
  mse_right = with(right, sum((y - mean(y)) ^ 2))

  mse_left + mse_right # also consider mae

}

# calculate mse for each possible split, find best
which.min(purrr::map_dbl(mids, calc_mse_split, df = df))
```

```
## [1] 55
```

Chapter 10

Bias–Variance Tradeoff

Consider the general regression setup where we are given a random pair $(X, Y) \in \mathbb{R}^p \times \mathbb{R}$. We would like to "predict" Y with some function of X, say, $f(X)$.

To clarify what we mean by "predict," we specify that we would like $f(X)$ to be "close" to Y. To further clarify what we mean by "close," we define the **squared error loss** of estimating Y using $f(X)$.

$$L(Y, f(X)) \triangleq (Y - f(X))^2$$

Now we can clarify the goal of regression, which is to minimize the above loss, on average. We call this the **risk** of estimating Y using $f(X)$.

$$R(Y, f(X)) \triangleq \mathbb{E}[L(Y, f(X))] = \mathbb{E}_{X,Y}[(Y - f(X))^2]$$

Before attempting to minimize the risk, we first re-write the risk after conditioning on X.

$$\mathbb{E}_{X,Y}\left[(Y - f(X))^2\right] = \mathbb{E}_X \mathbb{E}_{Y|X}\left[(Y - f(X))^2 \mid X = x\right]$$

Minimizing the right-hand side is much easier, as it simply amounts to minimizing the inner expectation with respect to $Y \mid X$, essentially minimizing the risk pointwise, for each x.

It turns out, that the risk is minimized by the conditional mean of Y given X,

$$f(x) = \mathbb{E}(Y \mid X = x)$$

which we call the **regression function**.

Note that the choice of squared error loss is somewhat arbitrary. Suppose instead we chose absolute error loss.

$$L(Y, f(X)) \triangleq |Y - f(X)|$$

The risk would then be minimized by the conditional median.

$$f(x) = \text{median}(Y \mid X = x)$$

Despite this possibility, our preference will still be for squared error loss. The reasons for this are numerous, including: historical, ease of optimization, and protecting against large deviations.

Now, given data $\mathcal{D} = (x_i, y_i) \in \mathbb{R}^p \times \mathbb{R}$, our goal becomes finding some $\hat{f}$ that is a good estimate of the regression function f. We'll see that this amounts to minimizing what we call the reducible error.

10.1 Reducible and Irreducible Error

Suppose that we obtain some $\hat{f}$, how well does it estimate f? We define the **expected prediction error** of predicting Y using $\hat{f}(X)$. A good $\hat{f}$ will have a low expected prediction error.

$$\text{EPE}\left(Y, \hat{f}(X)\right) \triangleq \mathbb{E}_{X,Y,\mathcal{D}}\left[\left(Y - \hat{f}(X)\right)^2\right]$$

This expectation is over X, Y, and also $\mathcal{D}$. The estimate $\hat{f}$ is actually random depending on the sampled data $\mathcal{D}$. We could actually write $\hat{f}(X, \mathcal{D})$ to make this dependence explicit, but our notation will become cumbersome enough as it is.

Like before, we'll condition on X. This results in the expected prediction error of predicting Y using $\hat{f}(X)$ when $X = x$.

$$\text{EPE}\left(Y, \hat{f}(x)\right) = \mathbb{E}_{Y|X,\mathcal{D}}\left[\left(Y - \hat{f}(X)\right)^2 \mid X = x\right] = \underbrace{\mathbb{E}_{\mathcal{D}}\left[\left(f(x) - \hat{f}(x)\right)^2\right]}_{\text{reducible error}} + \underbrace{\mathbb{V}_{Y|X}\left[Y \mid X = x\right]}_{\text{irreducible error}}$$

A number of things to note here:

- The expected prediction error is for a random Y given a fixed x and a random $\hat{f}$. As such, the expectation is over $Y \mid X$ and $\mathcal{D}$. Our estimated function $\hat{f}$ is random depending on the sampled data, $\mathcal{D}$, which is used to perform the estimation.
- The expected prediction error of predicting Y using $\hat{f}(X)$ when $X = x$ has been decomposed into two errors:
  - The **reducible error**, which is the expected squared error loss of estimation $f(x)$ using $\hat{f}(x)$ at a fixed point x. The only thing that is random here is $\mathcal{D}$, the data used to obtain $\hat{f}$. (Both f and x are fixed.) We'll often call this reducible error the **mean squared error** of estimating $f(x)$ using $\hat{f}$ at a fixed point x.

$$\text{MSE}\left(f(x), \hat{f}(x)\right) \triangleq \mathbb{E}_{\mathcal{D}}\left[\left(f(x) - \hat{f}(x)\right)^2\right]$$

  - The **irreducible error**. This is simply the variance of Y given that $X = x$, essentially noise that we do not want to learn. This is also called the **Bayes error**.

As the name suggests, the reducible error is the error that we have some control over. But how do we control this error?

10.2 Bias-Variance Decomposition

After decomposing the expected prediction error into reducible and irreducible error, we can further decompose the reducible error.

Recall the definition of the **bias** of an estimator.

$$\text{bias}(\hat{\theta}) \triangleq \mathbb{E}\left[\hat{\theta}\right] - \theta$$

Also recall the definition of the **variance** of an estimator.

$$\mathbb{V}(\hat{\theta}) = \text{var}(\hat{\theta}) \triangleq \mathbb{E}\left[(\hat{\theta} - \mathbb{E}\left[\hat{\theta}\right])^2\right]$$

Using this, we further decompose the reducible error (mean squared error) into bias squared and variance.

$$\text{MSE}\left(f(x), \hat{f}(x)\right) = \mathbb{E}_{\mathcal{D}}\left[\left(f(x) - \hat{f}(x)\right)^2\right] = \underbrace{\left(f(x) - \mathbb{E}\left[\hat{f}(x)\right]\right)^2}_{\text{bias}^2\left(\hat{f}(x)\right)} + \underbrace{\mathbb{E}\left[\left(\hat{f}(x) - \mathbb{E}\left[\hat{f}(x)\right]\right)^2\right]}_{\text{var}\left(\hat{f}(x)\right)}$$

This is actually a common fact in estimation theory, but we have stated it here specifically for estimation of some regression function f using $\hat{f}$ at some point x.

$$\text{MSE}\left(f(x), \hat{f}(x)\right) = \text{bias}^2\left(\hat{f}(x)\right) + \text{var}\left(\hat{f}(x)\right)$$

In a perfect world, we would be able to find some $\hat{f}$ which is **unbiased**, that is $\text{bias}\left(\hat{f}(x)\right) = 0$, which also has low variance. In practice, this isn't always possible.

It turns out, there is a **bias-variance tradeoff**. That is, often, the more bias in our estimation, the lesser the variance. Similarly, less variance is often accompanied by more bias. Complex models tend to be unbiased, but highly variable. Simple models are often extremely biased, but have low variance.

In the context of regression, models are biased when:

- Parametric: The form of the model does not incorporate all the necessary variables, or the form of the relationship is too simple. For example, a parametric model assumes a linear relationship, but the true relationship is quadratic.
- Non-parametric: The model provides too much smoothing.

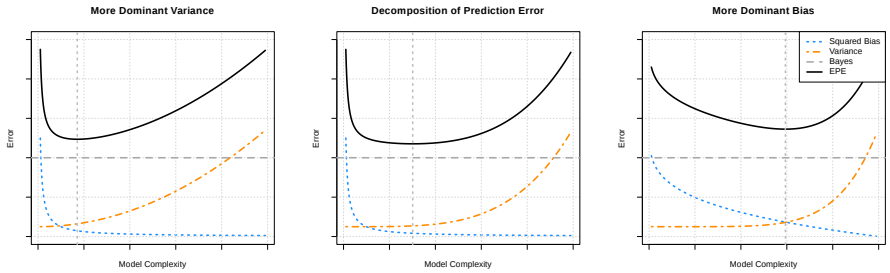
In the context of regression, models are variable when:

- Parametric: The form of the model incorporates too many variables, or the form of the relationship is too complex. For example, a parametric model assumes a cubic relationship, but the true relationship is linear.
- Non-parametric: The model does not provide enough smoothing. It is very, "wiggly."

So for us, to select a model that appropriately balances the tradeoff between bias and variance, and thus minimizes the reducible error, we need to select a model of the appropriate complexity for the data.

Recall that when fitting models, we've seen that train RMSE decreases as model complexity is increasing. (Technically it is non-increasing.) For test RMSE, we expect to see a U-shaped curve. Importantly, test RMSE decreases, until a certain complexity, then begins to increase.

Now we can understand why this is happening. The expected test RMSE is essentially the expected prediction error, which we now known decomposes into (squared) bias, variance, and the irreducible Bayes error. The following plots show three examples of this.



The three plots show three examples of the bias-variance tradeoff. In the left panel, the variance influences the expected prediction error more than the bias. In the right panel, the opposite is true. The middle panel is somewhat neutral. In all cases, the difference between the Bayes error (the horizontal dashed grey line) and the expected prediction error (the solid black curve) is exactly the mean squared error, which is the sum of the squared bias (blue curve) and variance (orange curve). The vertical line indicates the complexity that minimizes the prediction error.

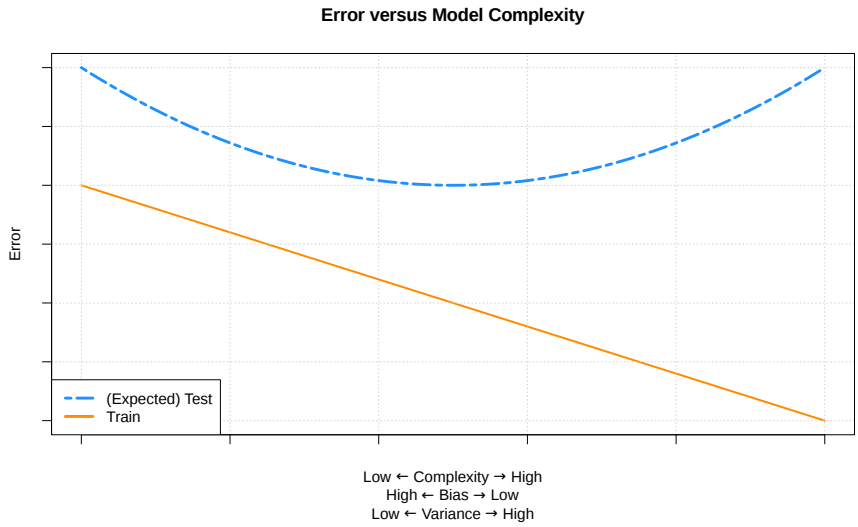
To summarize, if we assume that irreducible error can be written as

$$\mathbb{V}[Y \mid X = x] = \sigma^2$$

then we can write the full decomposition of the expected prediction error of predicting Y using $\hat{f}$ when $X = x$ as

$$\text{EPE}\left(Y, \hat{f}(x)\right) = \underbrace{\text{bias}^2\left(\hat{f}(x)\right) + \text{var}\left(\hat{f}(x)\right)}_{\text{reducible error}} + \sigma^2.$$

As model complexity increases, bias decreases, while variance increases. By understanding the tradeoff between bias and variance, we can manipulate model complexity to find a model that well predict well on unseen observations.



10.3 Simulation

We will illustrate these decompositions, most importantly the bias-variance tradeoff, through simulation. Suppose we would like to train a model to learn the true regression function function $f(x) = x^2$.

```
f = function(x) {
  x ^ 2
}
```

More specifically, we'd like to predict an observation, Y, given that $X = x$ by using $\hat{f}(x)$ where

$$\mathbb{E}[Y \mid X = x] = f(x) = x^2$$

and

$$\mathbb{V}[Y \mid X = x] = \sigma^2.$$

Alternatively, we could write this as

$$Y = f(X) + \epsilon$$

where $\mathbb{E}[\epsilon] = 0$ and $\mathbb{V}[\epsilon] = \sigma^2$. In this formulation, we call $f(X)$ the **signal** and ϵ the **noise**.

To carry out a concrete simulation example, we need to fully specify the data generating process. We do so with the following R code.

```
gen_sim_data = function(f, sample_size = 100) {
  x = runif(n = sample_size, min = 0, max = 1)
  y = rnorm(n = sample_size, mean = f(x), sd = 0.3)
  data.frame(x, y)
}
```

Also note that if you prefer to think of this situation using the $Y = f(X) + \epsilon$ formulation, the following code represents the same data generating process.

```
gen_sim_data = function(f, sample_size = 100) {
  x = runif(n = sample_size, min = 0, max = 1)
  eps = rnorm(n = sample_size, mean = 0, sd = 0.75)
  y = f(x) + eps
  data.frame(x, y)
}
```

To completely specify the data generating process, we have made more model assumptions than simply $\mathbb{E}[Y \mid X = x] = x^2$ and $\mathbb{V}[Y \mid X = x] = \sigma^2$. In particular,

- The x_i in $\mathcal{D}$ are sampled from a uniform distribution over $[0, 1]$.
- The x_i and ϵ are independent.
- The y_i in $\mathcal{D}$ are sampled from the conditional normal distribution.

$$Y \mid X \sim N(f(x), \sigma^2)$$

Using this setup, we will generate datasets, $\mathcal{D}$, with a sample size $n = 100$ and fit four models.

$$\begin{aligned}
\texttt{predict(fit0, x)} &= \hat{f}_0(x) = \hat{\beta}_0 \\
\texttt{predict(fit1, x)} &= \hat{f}_1(x) = \hat{\beta}_0 + \hat{\beta}_1 x \\
\texttt{predict(fit2, x)} &= \hat{f}_2(x) = \hat{\beta}_0 + \hat{\beta}_1 x + \hat{\beta}_2 x^2 \\
\texttt{predict(fit9, x)} &= \hat{f}_9(x) = \hat{\beta}_0 + \hat{\beta}_1 x + \hat{\beta}_2 x^2 + \ldots + \hat{\beta}_9 x^9
\end{aligned}$$

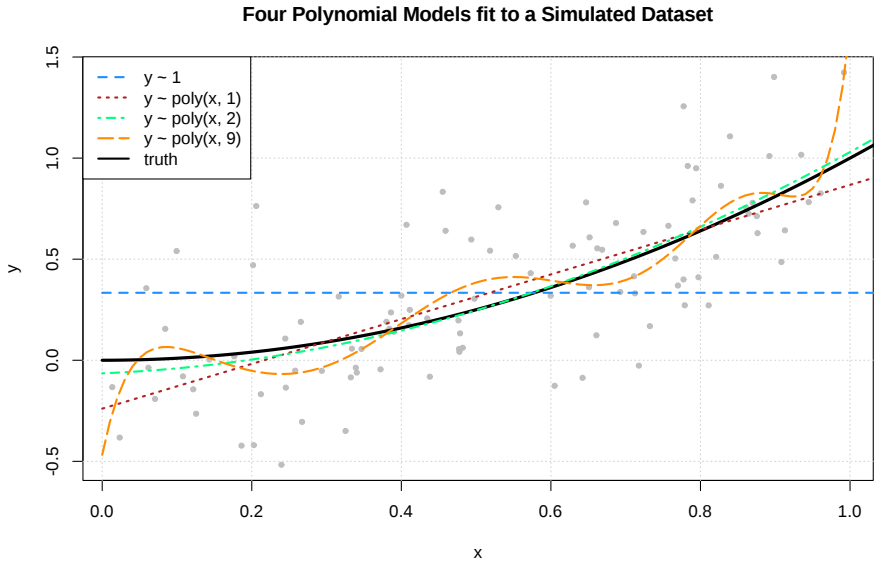
To get a sense of the data and these four models, we generate one simulated dataset, and fit the four models.

```
set.seed(1)
sim_data = gen_sim_data(f)
```

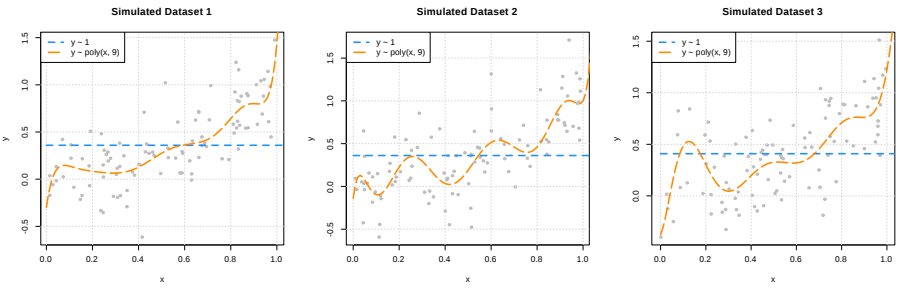
```
fit_0 = lm(y ~ 1,                   data = sim_data)
fit_1 = lm(y ~ poly(x, degree = 1), data = sim_data)
fit_2 = lm(y ~ poly(x, degree = 2), data = sim_data)
fit_9 = lm(y ~ poly(x, degree = 9), data = sim_data)
```

Note that technically we're being lazy and using orthogonal polynomials, but the fitted values are the same, so this makes no difference for our purposes.

Plotting these four trained models, we see that the zero predictor model does very poorly. The first degree model is reasonable, but we can see that the second degree model fits much better. The ninth degree model seem rather wild.



The following three plots were created using three additional simulated datasets. The zero predictor and ninth degree polynomial were fit to each.

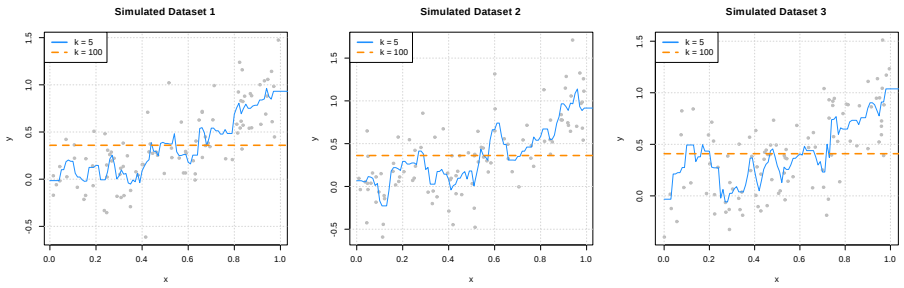


This plot should make clear the difference between the bias and variance of these two models. The zero predictor model is clearly wrong, that is, biased, but nearly the same for each of the datasets, since it has very low variance.

While the ninth degree model doesn't appear to be correct for any of these three

simulations, we'll see that on average it is, and thus is performing unbiased estimation. These plots do however clearly illustrate that the ninth degree polynomial is extremely variable. Each dataset results in a very different fitted model. Correct on average isn't the only goal we're after, since in practice, we'll only have a single dataset. This is why we'd also like our models to exhibit low variance.

We could have also fit k-nearest neighbors models to these three datasets.



Here we see that when $k = 100$ we have a biased model with very low variance. (It's actually the same as the 0 predictor linear model.) When $k = 5$, we again have a highly variable model.

These two sets of plots reinforce our intuition about the bias-variance tradeoff. Complex models (ninth degree polynomial and $k = 5$) are highly variable, and often unbiased. Simple models (zero predictor linear model and $k = 100$) are very biased, but have extremely low variance.

We will now complete a simulation study to understand the relationship between the bias, variance, and mean squared error for the estimates for $f(x)$ given by these four models at the point $x = 0.90$. We use simulation to complete this task, as performing the analytical calculations would prove to be rather tedious and difficult.

```
set.seed(1)
n_sims = 250
n_models = 4
x = data.frame(x = 0.90) # fixed point at which we make predictions
predictions = matrix(0, nrow = n_sims, ncol = n_models)
```

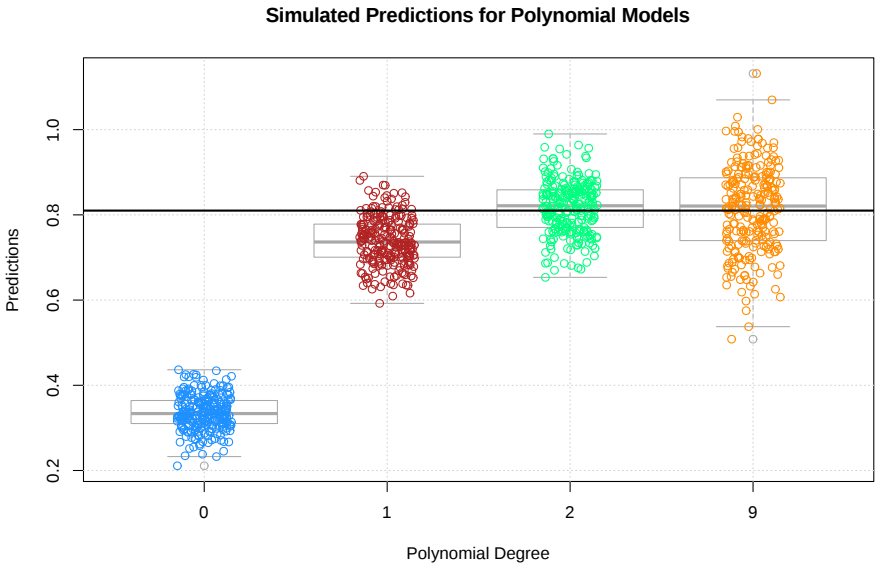
```
for (sim in 1:n_sims) {

  # simulate new, random, training data
  # this is the only random portion of the bias, var, and mse calculations
  # this allows us to calculate the expectation over D
  sim_data = gen_sim_data(f)

  # fit models
```

```
  fit_0 = lm(y ~ 1,                    data = sim_data)
  fit_1 = lm(y ~ poly(x, degree = 1), data = sim_data)
  fit_2 = lm(y ~ poly(x, degree = 2), data = sim_data)
  fit_9 = lm(y ~ poly(x, degree = 9), data = sim_data)

  # get predictions
  predictions[sim, 1] = predict(fit_0, x)
  predictions[sim, 2] = predict(fit_1, x)
  predictions[sim, 3] = predict(fit_2, x)
  predictions[sim, 4] = predict(fit_9, x)
}
```

Note that this is one of many ways we could have accomplished this task using `R`. For example we could have used a combination of `replicate()` and `*apply()` functions. Alternatively, we could have used a `tidyverse` approach, which likely would have used some combination of `dplyr`, `tidyr`, and `purrr`.

Our approach, which would be considered a `base R` approach, was chosen to make it as clear as possible what is being done. The `tidyverse` approach is rapidly gaining popularity in the `R` community, but might make it more difficult to see what is happening here, unless you are already familiar with that approach.

Also of note, while it may seem like the output stored in `predictions` would meet the definition of tidy data given by Hadley Wickham since each row represents a simulation, it actually falls slightly short. For our data to be tidy, a row should store the simulation number, the model, and the resulting prediction. We've actually already aggregated one level above this. Our observational unit is a simulation (with four predictions), but for tidy data, it should be a single prediction. This may be revised by the author later when there are more examples of how to do this from the `R` community.



The above plot shows the predictions for each of the 250 simulations of each of the four models of different polynomial degrees. The truth, $f(x = 0.90) = (0.9)^2 = 0.81$, is given by the solid black horizontal line.

Two things are immediately clear:

- As complexity *increases*, **bias decreases**. (The mean of a model's predictions is closer to the truth.)
- As complexity *increases*, **variance increases**. (The variance about the mean of a model's predictions increases.)

The goal of this simulation study is to show that the following holds true for each of the four models.

$$\text{MSE}\left(f(0.90), \hat{f}_k(0.90)\right) = \underbrace{\left(\mathbb{E}\left[\hat{f}_k(0.90)\right] - f(0.90)\right)^2}_{\text{bias}^2\left(\hat{f}_k(0.90)\right)} + \underbrace{\mathbb{E}\left[\left(\hat{f}_k(0.90) - \mathbb{E}\left[\hat{f}_k(0.90)\right]\right)^2\right]}_{\text{var}\left(\hat{f}_k(0.90)\right)}$$

We'll use the empirical results of our simulations to estimate these quantities. (Yes, we're using estimation to justify facts about estimation.) Note that we've actually used a rather small number of simulations. In practice we should use more, but for the sake of computation time, we've performed just enough simulations to obtain the desired results. (Since we're estimating estimation, the bigger the sample size, the better.)

To estimate the mean squared error of our predictions, we'll use

$$\widehat{\text{MSE}}\left(f(0.90), \hat{f}_k(0.90)\right) = \frac{1}{n_{\texttt{sims}}} \sum_{i=1}^{n_{\texttt{sims}}} \left(f(0.90) - \hat{f}_k(0.90)\right)^2$$

We also write an accompanying `R` function.

```
get_mse = function(truth, estimate) {
  mean((estimate - truth) ^ 2)
}
```

Similarly, for the bias of our predictions we use,

$$\widehat{\text{bias}}\left(\hat{f}(0.90)\right) = \frac{1}{n_{\texttt{sims}}} \sum_{i=1}^{n_{\texttt{sims}}} \left(\hat{f}_k(0.90)\right) - f(0.90)$$

And again, we write an accompanying `R` function.

```
get_bias = function(estimate, truth) {
  mean(estimate) - truth
}
```

Lastly, for the variance of our predictions we have

$$\widehat{\text{var}}\left(\hat{f}(0.90)\right) = \frac{1}{n_{\texttt{sims}}} \sum_{i=1}^{n_{\texttt{sims}}} \left(\hat{f}_k(0.90) - \frac{1}{n_{\texttt{sims}}} \sum_{i=1}^{n_{\texttt{sims}}} \hat{f}_k(0.90)\right)^2$$

While there is already `R` function for variance, the following is more appropriate in this situation.

```
get_var = function(estimate) {
  mean((estimate - mean(estimate)) ^ 2)
}
```

To quickly obtain these results for each of the four models, we utilize the `apply()` function.

```
bias = apply(predictions, 2, get_bias, truth = f(x = 0.90))
variance = apply(predictions, 2, get_var)
mse = apply(predictions, 2, get_mse, truth = f(x = 0.90))
```

We summarize these results in the following table.

Degree	Mean Squared Error	Bias Squared	Variance
0	0.22643	0.22476	0.00167
1	0.00829	0.00508	0.00322
2	0.00387	0.00005	0.00381
9	0.01019	0.00002	0.01017

A number of things to notice here:

- We use squared bias in this table. Since bias can be positive or negative, squared bias is more useful for observing the trend as complexity increases.
- The squared bias trend which we see here is **decreasing** as complexity increases, which we expect to see in general.
- The exact opposite is true of variance. As model complexity increases, variance **increases**.
- The mean squared error, which is a function of the bias and variance, decreases, then increases. This is a result of the bias-variance tradeoff. We can decrease bias, by increasing variance. Or, we can decrease variance by increasing bias. By striking the correct balance, we can find a good mean squared error!

We can check for these trends with the `diff()` function in `R`.

```
all(diff(bias ^ 2) < 0)
```

```
## [1] TRUE
```

```
all(diff(variance) > 0)
```

```
## [1] TRUE
```

```
diff(mse) < 0
```

```
##     1     2     9
##  TRUE  TRUE FALSE
```

The models with polynomial degrees 2 and 9 are both essentially unbiased. We see some bias here as a result of using simulation. If we increased the number of simulations, we would see both biases go down. Since they are both unbiased, the model with degree 2 outperforms the model with degree 9 due to its smaller variance.

Models with degree 0 and 1 are biased because they assume the wrong form of the regression function. While the degree 9 model does this as well, it does include all the necessary polynomial degrees.

$$\hat{f}_9(x) = \hat{\beta}_0 + \hat{\beta}_1 x + \hat{\beta}_2 x^2 + \ldots + \hat{\beta}_9 x^9$$

Then, since least squares estimation is unbiased, importantly,

$$\mathbb{E}[\hat{\beta}_d] = \beta_d = 0$$

for $d = 3, 4, \ldots 9$, we have

$$\mathbb{E}\left[\hat{f}_9(x)\right] = \beta_0 + \beta_1 x + \beta_2 x^2$$

Now we can finally verify the bias-variance decomposition.

```
bias ^ 2 + variance == mse
```

```
##     0     1     2     9
## FALSE FALSE FALSE  TRUE
```

But wait, this says it isn't true, except for the degree 9 model? It turns out, this is simply a computational issue. If we allow for some very small error tolerance, we see that the bias-variance decomposition is indeed true for predictions from these for models.

```
all.equal(bias ^ 2 + variance, mse)
```

```
## [1] TRUE
```

See `?all.equal()` for details.

So far, we've focused our efforts on looking at the mean squared error of estimating $f(0.90)$ using $\hat{f}(0.90)$. We could also look at the expected prediction error of using $\hat{f}(X)$ when $X = 0.90$ to estimate Y.

$$\text{EPE}\left(Y, \hat{f}_k(0.90)\right) = \mathbb{E}_{Y|X,\mathcal{D}}\left[\left(Y - \hat{f}_k(X)\right)^2 \mid X = 0.90\right]$$

We can estimate this quantity for each of the four models using the simulation study we already performed.

```
get_epe = function(realized, estimate) {
  mean((realized - estimate) ^ 2)
}
```

```
y = rnorm(n = nrow(predictions), mean = f(x = 0.9), sd = 0.3)
epe = apply(predictions, 2, get_epe, realized = y)
epe
```

```
##         0         1         2         9
## 0.3180470 0.1104055 0.1095955 0.1205570
```

What about the unconditional expected prediction error. That is, for any X, not just 0.90. Specifically, the expected prediction error of estimating Y using $\hat{f}(X)$. The following (new) simulation study provides an estimate of

$$\text{EPE}\left(Y, \hat{f}_k(X)\right) = \mathbb{E}_{X,Y,\mathcal{D}}\left[\left(Y - \hat{f}_k(X)\right)^2\right]$$

for the quadratic model, that is $k = 2$ as we have defined k.

```
set.seed(1)
n_sims = 1000
```

```
X = runif(n = n_sims, min = 0, max = 1)
Y = rnorm(n = n_sims, mean = f(X), sd = 0.3)

f_hat_X = rep(0, length(X))

for (i in seq_along(X)) {
  sim_data = gen_sim_data(f)
  fit_2 = lm(y ~ poly(x, degree = 2), data = sim_data)
  f_hat_X[i] = predict(fit_2, newdata = data.frame(x = X[i]))
}

mean((Y - f_hat_X) ^ 2)
```

```
## [1] 0.09997319
```

Note that in practice, we should use many more simulations in this study.

10.4 Estimating Expected Prediction Error

While previously, we only decomposed the expected prediction error conditionally, a similar argument holds unconditionally.

Assuming

$$\mathbb{V}[Y \mid X = x] = \sigma^2.$$

we have

$$\text{EPE}\left(Y, \hat{f}(X)\right) = \mathbb{E}_{X,Y,\mathcal{D}}\left[(Y - \hat{f}(X))^2\right] = \underbrace{\mathbb{E}_X\left[\text{bias}^2\left(\hat{f}(X)\right)\right] + \mathbb{E}_X\left[\text{var}\left(\hat{f}(X)\right)\right]}_{\text{reducible error}} + \sigma^2$$

Lastly, we note that if

$$\mathcal{D} = \mathcal{D}_{\texttt{trn}} \cup \mathcal{D}_{\texttt{tst}} = (x_i, y_i) \in \mathbb{R}^p \times \mathbb{R}, \ i = 1, 2, \ldots n$$

where

$$\mathcal{D}_{\texttt{trn}} = (x_i, y_i) \in \mathbb{R}^p \times \mathbb{R}, \ i \in \texttt{trn}$$

and

$$\mathcal{D}_{\texttt{tst}} = (x_i, y_i) \in \mathbb{R}^p \times \mathbb{R}, \ i \in \texttt{tst}$$

Then, if we use $\mathcal{D}_{\texttt{trn}}$ to fit (train) a model, we can use the test mean squared error

$$\sum_{i \in \texttt{tst}} \left(y_i - \hat{f}(x_i)\right)^2$$

as an estimate of

$$\mathbb{E}_{X,Y,\mathcal{D}}\left[(Y - \hat{f}(X))^2\right]$$

the expected prediction error. (In practice we prefer RMSE to MSE for comparing models and reporting because of the units.)

How good is this estimate? Well, if $\mathcal{D}$ is a random sample from (X, Y), and `tst` are randomly sampled observations randomly sampled from $i = 1, 2, \ldots, n$, then it is a reasonable estimate. However, it is rather variable due to the randomness of selecting the observations for the test set. How variable? It turns out, pretty variable. While it's a justified estimate, eventually we'll introduce cross-validation as a procedure better suited to performing this estimation to select a model.

10.5 Reproducibility

The R Markdown file for this chapter can be found **here**. The file was created using `R` version 3.6.1.

Chapter 11

Classification

11.1 STAT 432 Materials

- **Slides** | Classification: Introduction
- **Code** | Some Classification Code
- **Slides** | Classification: Binary Classification
- **Code** | Some Binary Classification Code
- **Slides** | Classification: Nonparametric Classification
- **Reading** | STAT 420: Logistic Regression
- **Slides** | Classification: Logistic Regression

```
library("dplyr")
library("knitr")
library("kableExtra")
library("tibble")
library("caret")
library("rpart")
library("nnet")
```

11.2 Bayes Classifier

- TODO: Not the same as naïve Bayes classifier

$$p_k(x) = P\left[Y = k \mid X = x\right]$$

$$C^B(x) = \underset{k \in \{1,2,\ldots K\}}{\text{argmax}} \; P\left[Y = k \mid X = x\right]$$

11.2.1 Bayes Error Rate

$$1 - \mathbb{E}_X \left[\max_k P[Y = k \mid X = x] \right]$$

11.3 Building a Classifier

$$\hat{p}_k(x) = \hat{P}\left[Y = k \mid X = x\right]$$

$$\hat{C}(x) = \underset{k \in \{1,2,\ldots K\}}{\text{argmax}} \; \hat{p}_k(x)$$

- TODO: first estimation conditional distribution, then classify to label with highest probability

```
set.seed(1)
joint_probs = round(1:12 / sum(1:12), 2)
joint_probs = sample(joint_probs)
joint_dist = matrix(data  = joint_probs, nrow = 3, ncol = 4)
colnames(joint_dist) = c("$X = 1$", "$X = 2$", "$X = 3$", "$X = 4$")
rownames(joint_dist) = c("$Y = A$", "$Y = B$", "$Y = C$")
joint_dist %>%
  kable() %>%
  kable_styling("striped", full_width = FALSE) %>%
  column_spec(column = 1, bold = TRUE, background = "white", border_right = TRUE)
```

	$X = 1$	$X = 2$	$X = 3$	$X = 4$
$Y = A$	0.12	0.01	0.04	0.14
$Y = B$	0.05	0.03	0.10	0.15
$Y = C$	0.09	0.06	0.08	0.13

```
# marginal distribution of Y
t(colSums(joint_dist)) %>% kable() %>% kable_styling(full_width = FALSE)
```

$X = 1$	$X = 2$	$X = 3$	$X = 4$
0.26	0.1	0.22	0.42

```
# marginal distribution of X
t(rowSums(joint_dist)) %>% kable() %>% kable_styling(full_width = FALSE)
```

$Y = A$	$Y = B$	$Y = C$
0.31	0.33	0.36

```
gen_data = function(n = 100) {
  x = sample(c(0, 1), prob = c(0.4, 0.6), size = n, replace = TRUE)
  y = ifelse(test = {x == 0},
             yes = sample(c("A", "B", "C"), size = n, prob = c(0.25, 0.50, 0.25), replace = TRUE)
             no = sample(c("A", "B", "C"), size = n, prob = c(0.1, 0.1, 0.4) / 0.6, replace = TRU

  tibble(x = x, y = factor(y))
}

test_cases = tibble(x = c(0, 1))

set.seed(42)
some_data = gen_data()

predict(knn3(y ~ x, data = some_data), test_cases)
```

```
##               A          B         C
## [1,] 0.2608696 0.39130435 0.3478261
## [2,] 0.1481481 0.07407407 0.7777778
```

```
predict(rpart(y ~ x, data = some_data), test_cases)
```

```
##           A          B         C
## 1 0.2608696 0.39130435 0.3478261
## 2 0.1481481 0.07407407 0.7777778
```

```
predict(multinom(y ~ x, data = some_data, trace = FALSE), test_cases, type = "prob")
```

```
##           A          B         C
## 1 0.2608699 0.39130496 0.3478251
## 2 0.1481478 0.07407414 0.7777781
```

11.4 Modeling

11.4.1 Linear Models

- TODO: use `nnet::multinom`
  - in place of `glm()`? always?

11.4.2 k-Nearest Neighbors

- TODO: use `caret::knn3()`

11.4.3 Decision Trees

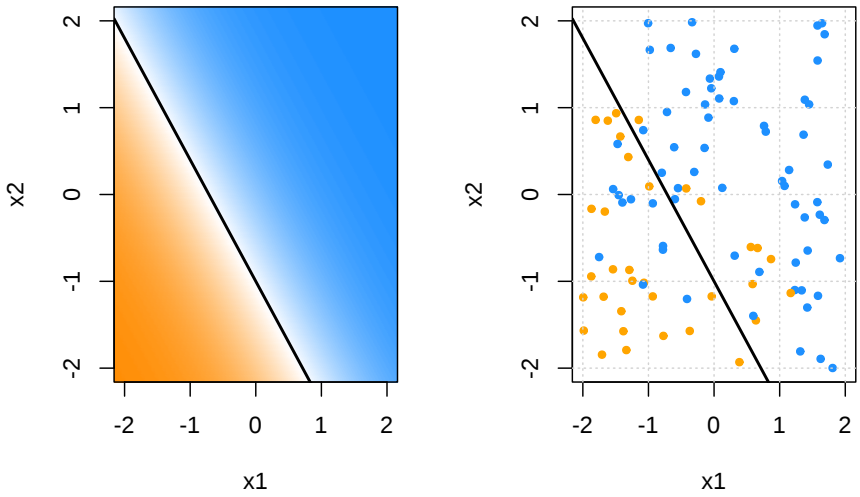
- TODO: use `rpart::rpart()`

11.5 MISC TODO STUFF

- TODO: https://topepo.github.io/caret/visualizations.html
- TODO: https://en.wikipedia.org/wiki/Confusion_matrix
- TODO: https://en.wikipedia.org/wiki/Matthews_correlation_coefficient
- TODO: https://people.inf.elte.hu/kiss/11dwhdm/roc.pdf
- TODO: https://www.cs.cmu.edu/~tom/mlbook/NBayesLogReg.pdf
- TODO: http://www.oranlooney.com/post/viz-tsne/
- TODO: https://web.expasy.org/pROC/
- TODO: https://bmcbioinformatics.biomedcentral.com/track/pdf/10.1186/1471-2105-12-77
- TODO: https://en.wikipedia.org/wiki/Receiver_operating_characteristic
- TODO: https://papers.nips.cc/paper/2020-on-discriminative-vs-generative-classifiers-a-comparisc pdf
- https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.141.751&rep=rep1&type=pdf
- https://www.cs.ubc.ca/~murphyk/Teaching/CS340-Fall06/lectures/naiveBayes.pdf
- http://www.stat.cmu.edu/~ryantibs/statml/lectures/linearclassification.pdf
- https://www.cs.cmu.edu/~tom/mlbook/NBayesLogReg.pdf

```
sim_2d_logistic = function(beta_0, beta_1, beta_2, n) {

  par(mfrow = c(1, 2))

  prob_plane = as_tibble(expand.grid(x1 = -220:220 / 100,
```

```
                                    x2 = -220:220 / 100))
  prob_plane$p = with(prob_plane,
                      boot::inv.logit(beta_0 + beta_1 * x1 + beta_2 * x2))

  do_to_db = colorRampPalette(c('darkorange', "white", 'dodgerblue'))

  plot(x2 ~ x1, data = prob_plane,
       col = do_to_db(100)[as.numeric(cut(prob_plane$p,
                                          seq(0, 1, length.out = 101)))],
       xlim = c(-2, 2), ylim = c(-2, 2), pch = 20)
  abline(-beta_0 / beta_2, -beta_1 / beta_2, col = "black", lwd = 2)

  x1 = runif(n = n, -2, 2)
  x2 = runif(n = n, -2, 2)
  y = rbinom(n = n, size = 1, prob = boot::inv.logit(beta_0 + beta_1 * x1 + beta_2 * x2))
  y = ifelse(y == 1, "dodgerblue", "orange")
  asdf = tibble(x1, x2, y)

  plot(x2 ~ x1, data = asdf, col = y, xlim = c(-2, 2), ylim = c(-2, 2), pch = 20)
  grid()
  abline(-beta_0 / beta_2, -beta_1 / beta_2, col = "black", lwd = 2)

}

sim_2d_logistic(beta_0 = 2 * 0.5, beta_1 = 2* 0.7, beta_2 = 2* 0.5, n = 100)
```


x2
x1
-2
-1
0
1
2

Chapter 12

Resampling

12.1 STAT 432 Materials

- **Code** | Some Resampling Code

```
library("dplyr")
library("rsample")
library("tibble")
library("knitr")
library("kableExtra")
library("purrr")
```

In this chapter we introduce **cross-validation**. We will highlight the need for cross-validation by comparing it to our previous approach, which was to use a single **validation** set inside of the training data.

To illustrate the use of cross-validation, we'll consider a regression setup with a single feature x, and a regression function $f(x) = x^3$. Adding an additional noise parameter, and the distribution of the feature variable, we define the entire data generating process as

$$X \sim \text{Unif}\left(a = -1, b = 1\right) Y \mid X \sim \text{Normal}\left(\mu = x^3, \sigma^2 = 0.25^2\right)$$

We write an `R` function that generates datasets according to this process.

```
gen_sim_data = function(sample_size) {
  x = runif(n = sample_size, min = -1, max = 1)
  y = rnorm(n = sample_size, mean = x ^ 3, sd = 0.25)
```

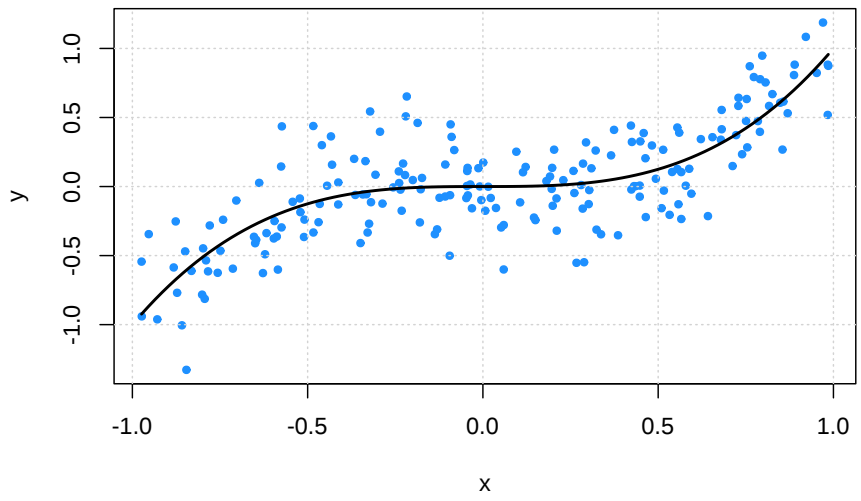
```
  tibble(x, y)
}
```

We first simulate a single train dataset, which we also split into an *estimation* and *validation* set. We also simulate a large test dataset. (Which we could not do in pratice, but is possible here.)

```
set.seed(1)
sim_trn = gen_sim_data(sample_size = 200)
sim_idx = sample(1:nrow(sim_trn), 160)
sim_est = sim_trn[sim_idx, ]
sim_val = sim_trn[-sim_idx, ]
sim_tst = gen_sim_data(sample_size = 10000)
```

We plot this training data, as well as the true regression function.

```
plot(y ~ x, data = sim_trn, col = "dodgerblue", pch = 20)
grid()
curve(x ^ 3, add = TRUE, col = "black", lwd = 2)
```




```
calc_rmse = function(actual, predicted) {
  sqrt(mean((actual - predicted) ^ 2))
}
```

Recall that we needed this validation set because the training error was far too optimistic for highly flexible models. This would lead us to always use the most flexible model. (That is, data that is used to fit a model should not be used to

validate a model.)

```
tibble(
  "Polynomial Degree" = 1:10,
  "Train RMSE" = map_dbl(1:10, ~ calc_rmse(actual = sim_est$y, predicted = predict(lm(y ~ poly(x,
  "Validation RMSE" = map_dbl(1:10, ~ calc_rmse(actual = sim_val$y, predicted = predict(lm(y ~ po
) %>%
  kable(digits = 4) %>%
  kable_styling("striped", full_width = FALSE)
```

Polynomial Degree	Train RMSE	Validation RMSE
1	0.2865	0.3233
2	0.2861	0.3220
3	0.2400	0.2746
4	0.2398	0.2754
5	0.2288	0.2832
6	0.2288	0.2833
7	0.2287	0.2820
8	0.2286	0.2805
9	0.2286	0.2803
10	0.2267	0.2886

12.2 Validation-Set Approach

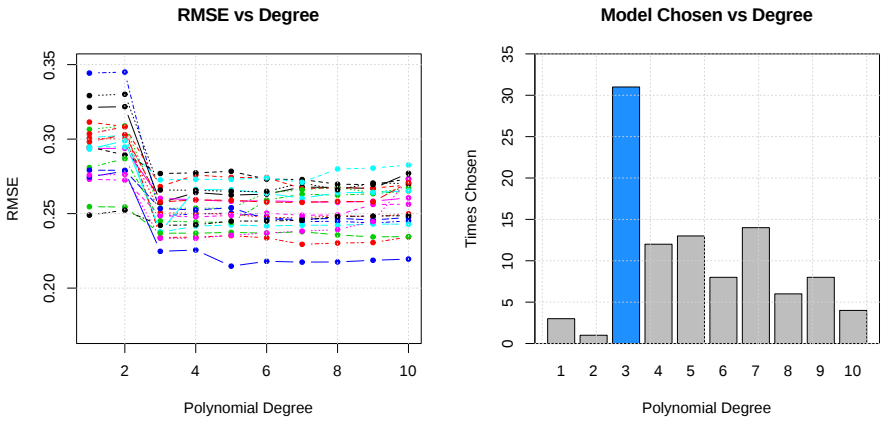
- TODO: consider fitting polynomial models of degree k = 1:10 to data from this data generating process
- TODO: here, we can consider k, the polynomial degree, as a tuning parameter
- TODO: perform simulation study to evaluate how well validation set approach works

```
num_sims = 100
num_degrees = 10
val_rmse = matrix(0, ncol = num_degrees, nrow = num_sims)
```

- TODO: each simulation we will...

```
set.seed(42)
for (i in 1:num_sims) {

  # simulate data
  sim_trn = gen_sim_data(sample_size = 200)

  # set aside validation set
  sim_idx = sample(1:nrow(sim_trn), 160)
```

```
  sim_est = sim_trn[sim_idx, ]
  sim_val = sim_trn[-sim_idx, ]

  # fit models and store RMSEs
  for (j in 1:num_degrees) {

    #fit model
    fit = glm(y ~ poly(x, degree = j), data = sim_est)

    # calculate error
    val_rmse[i, j] = calc_rmse(actual = sim_val$y, predicted = predict(fit, sim_val))
  }
}
```



- TODO: issues are hard to "see" but have to do with variability
- TODO: sometimes we are selecting models that are not flexible enough!

12.3 Cross-Validation

Instead of using a single estimation-validation split, we instead look to use K-fold cross-validation.

$$\text{RMSE-CV}_K = \sum_{k=1}^{K} \frac{n_k}{n} \text{RMSE}_k$$

$$\text{RMSE}_k = \sqrt{\frac{1}{n_k} \sum_{i \in C_k} \left(y_i - \hat{f}^{-k}(x_i) \right)^2}$$

- n_k is the number of observations in fold k
- C_k are the observations in fold k
- $\hat{f}^{-k}()$ is the trained model using the training data without fold k

If n_k is the same in each fold, then

$$\text{RMSE-CV}_K = \frac{1}{K}\sum_{k=1}^{K}\text{RMSE}_k$$

- TODO: create and add graphic that shows the splitting process
- TODO: Can be used with any metric, MSE, RMSE, class-err, class-acc

There are many ways to perform cross-validation in `R`, depending on the statistical learning method of interest. Some methods, for example `glm()` through `boot::cv.glm()` and `knn()` through `knn.cv()` have cross-validation capabilities built-in. We'll use `glm()` for illustration. First we need to convince ourselves that `glm()` can be used to perform the same tasks as `lm()`.

```
glm_fit = glm(y ~ poly(x, 3), data = sim_trn)
coef(glm_fit)
```

```
## (Intercept) poly(x, 3)1 poly(x, 3)2 poly(x, 3)3
## -0.02516901  5.06661745 -0.09349681  2.64581436
```

```
lm_fit  = lm(y ~ poly(x, 3), data = sim_trn)
coef(lm_fit)
```

```
## (Intercept) poly(x, 3)1 poly(x, 3)2 poly(x, 3)3
## -0.02516901  5.06661745 -0.09349681  2.64581436
```

By default, `cv.glm()` will report leave-one-out cross-validation (LOOCV).

```
sqrt(boot::cv.glm(sim_trn, glm_fit)$delta)
```

```
## [1] 0.2488233 0.2488099
```

We are actually given two values. The first is exactly the LOOCV-MSE. The second is a minor correction that we will not worry about. We take a square root to obtain LOOCV-RMSE.

In practice, we often prefer 5 or 10-fold cross-validation for a number of reason, but often most importantly, for computational efficiency.

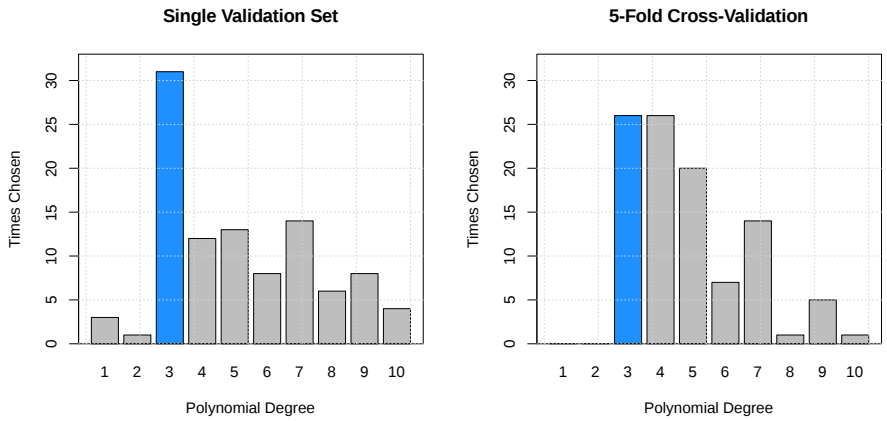
```
sqrt(boot::cv.glm(sim_trn, glm_fit, K = 5)$delta)
```

```
## [1] 0.2470322 0.2466417
```

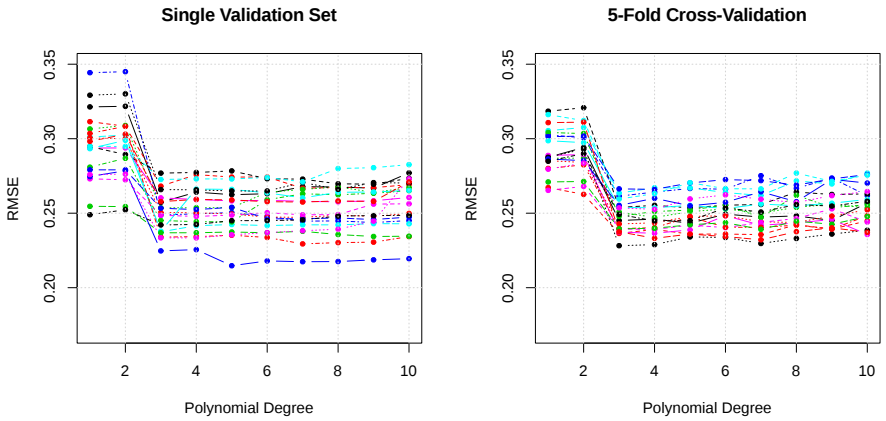
We repeat the above simulation study, this time performing 5-fold cross-validation. With a total sample size of $n = 200$ each validation set has 40 observations, as did the single validation set in the previous simulations.

```
cv_rmse = matrix(0, ncol = num_degrees, nrow = num_sims)
```

```
set.seed(42)
for (i in 1:num_sims) {

  # simulate data, use all data for training
  sim_trn = gen_sim_data(sample_size = 200)

  # fit models and store RMSE
  for (j in 1:num_degrees) {

    #fit model
    fit = glm(y ~ poly(x, degree = j), data = sim_trn)

    # calculate error
    cv_rmse[i, j] = sqrt(boot::cv.glm(sim_trn, fit, K = 5)$delta[1])
  }
}
```



Polynomial Degree	Mean, Val	SD, Val	Mean, CV	SD, CV
1	0.290	0.031	0.293	0.015
2	0.291	0.031	0.295	0.014
3	0.247	0.027	0.251	0.010
4	0.248	0.028	0.252	0.010
5	0.248	0.027	0.253	0.010
6	0.249	0.027	0.254	0.011
7	0.251	0.027	0.255	0.012
8	0.252	0.027	0.257	0.011
9	0.253	0.028	0.258	0.012
10	0.255	0.027	0.259	0.012



- TODO: differences: less variance, better selections

12.4 Test Data

The following example, inspired by The Elements of Statistical Learning, will illustrate the need for a dedicated test set which is **never** used in model training. We do this, if for no other reason, because it gives us a quick sanity check that we have cross-validated correctly. To be specific we will always test-train split the data, then perform cross-validation **within the training data**.

Essentially, this example will also show how to **not** cross-validate properly. It will also show an example of cross-validation in a classification setting.

```
calc_misclass = function(actual, predicted) {
  mean(actual != predicted)
}
```

Consider a binary response Y with equal probability to take values 0 and 1.

$$Y \sim \text{bern}(p = 0.5)$$

Also consider $p = 10,000$ independent predictor variables, X_j, each with a standard normal distribution.

$$X_j \sim N(\mu = 0, \sigma^2 = 1)$$

We simulate $n = 100$ observations from this data generating process. Notice that the way we've defined this process, none of the X_j are related to Y.

```
set.seed(42)
n = 200
p = 10000
x = replicate(p, rnorm(n))
y = c(rbinom(n = n, size = 1, prob = 0.5))
full_data = as_tibble(data.frame(y, x))
full_data
```

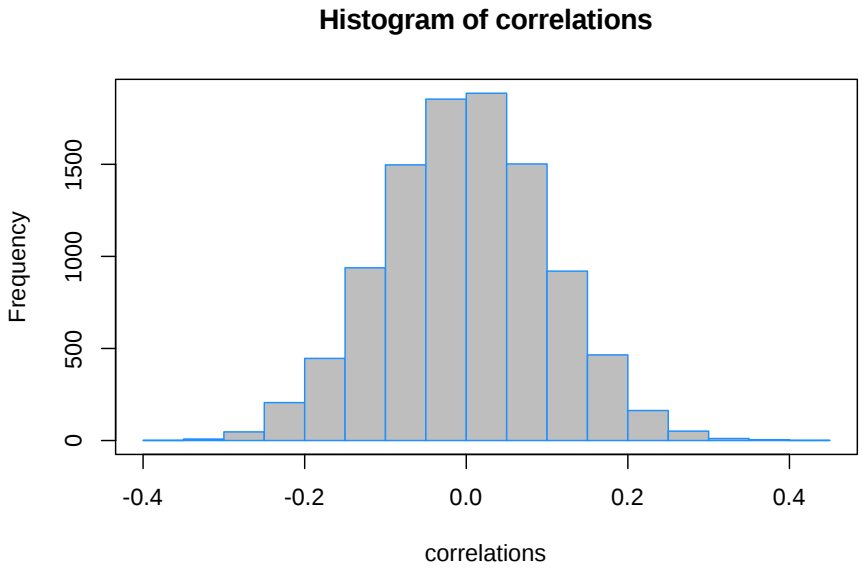
```
## # A tibble: 200 x 10,001
##        y      X1     X2       X3     X4     X5     X6      X7      X8      X9
##    <int>   <dbl>  <dbl>    <dbl>  <dbl>  <dbl>  <dbl>   <dbl>   <dbl>   <dbl>
##  1     1  1.37   -2.00   1.33    -0.248  0.689  2.33  -0.747   0.877   0.0293
##  2     1 -0.565   0.334 -0.869    0.422  0.725  0.524  0.0366 -1.77    1.93
##  3     1  0.363   1.17   0.0555   0.988  0.217  0.971  0.323  -0.0457  0.838
##  4     0  0.633   2.06   0.0491   0.836 -0.202  0.377  0.380  -0.395  -0.185
##  5     0  0.404  -1.38  -0.578   -0.661 -1.37  -0.996  0.877  -0.128   0.533
##  6     1 -0.106  -1.15  -0.999    1.56  -0.309 -0.597  0.933   1.10    1.78
##  7     1  1.51   -0.706 -0.00243 -1.62  -0.453  0.165 -2.43   -1.26    0.190
##  8     0 -0.0947 -1.05   0.656    0.864  0.663 -2.93   1.73   -0.265  -0.569
##  9     1  2.02   -0.646  1.48    -0.512  1.31  -0.848  0.456   2.55    1.45
## 10     0 -0.0627 -0.185 -1.91    -1.92   0.501  0.799 -0.570  -1.48    1.87
## # ... with 190 more rows, and 9,991 more variables: X10 <dbl>, X11 <dbl>,
## #   X12 <dbl>, X13 <dbl>, X14 <dbl>, X15 <dbl>, X16 <dbl>, X17 <dbl>,
## #   X18 <dbl>, X19 <dbl>, X20 <dbl>, X21 <dbl>, X22 <dbl>, X23 <dbl>,
## #   X24 <dbl>, X25 <dbl>, X26 <dbl>, X27 <dbl>, X28 <dbl>, X29 <dbl>,
## #   X30 <dbl>, X31 <dbl>, X32 <dbl>, X33 <dbl>, X34 <dbl>, X35 <dbl>,
## #   X36 <dbl>, X37 <dbl>, X38 <dbl>, X39 <dbl>, X40 <dbl>, X41 <dbl>,
## #   X42 <dbl>, X43 <dbl>, X44 <dbl>, X45 <dbl>, X46 <dbl>, X47 <dbl>,
## #   X48 <dbl>, X49 <dbl>, X50 <dbl>, X51 <dbl>, X52 <dbl>, X53 <dbl>,
## #   X54 <dbl>, X55 <dbl>, X56 <dbl>, X57 <dbl>, X58 <dbl>, X59 <dbl>,
## #   X60 <dbl>, X61 <dbl>, X62 <dbl>, X63 <dbl>, X64 <dbl>, X65 <dbl>,
## #   X66 <dbl>, X67 <dbl>, X68 <dbl>, X69 <dbl>, X70 <dbl>, X71 <dbl>,
## #   X72 <dbl>, X73 <dbl>, X74 <dbl>, X75 <dbl>, X76 <dbl>, X77 <dbl>,
## #   X78 <dbl>, X79 <dbl>, X80 <dbl>, X81 <dbl>, X82 <dbl>, X83 <dbl>,
## #   X84 <dbl>, X85 <dbl>, X86 <dbl>, X87 <dbl>, X88 <dbl>, X89 <dbl>,
## #   X90 <dbl>, X91 <dbl>, X92 <dbl>, X93 <dbl>, X94 <dbl>, X95 <dbl>,
## #   X96 <dbl>, X97 <dbl>, X98 <dbl>, X99 <dbl>, X100 <dbl>, X101 <dbl>,
## #   X102 <dbl>, X103 <dbl>, X104 <dbl>, X105 <dbl>, X106 <dbl>, X107 <dbl>,
## #   X108 <dbl>, X109 <dbl>, ...
```

Before attempting to perform cross-validation, we test-train split the data, using half of the available data for each. (In practice, with this little data, it would be hard to justify a separate test dataset, but here we do so to illustrate another point.)

```
trn_idx  = sample(1:nrow(full_data), trunc(nrow(full_data) * 0.5))
trn_data = full_data[trn_idx,   ]
tst_data = full_data[-trn_idx, ]
```

Now we would like to train a logistic regression model to predict Y using the available predictor data. However, here we have $p > n$, which prevents us from fitting logistic regression. To overcome this issue, we will first attempt to find a subset of relevant predictors. To do so, we'll simply find the predictors that are most correlated with the response.

```
# find correlation between y and each predictor variable
correlations = apply(trn_data[, -1], 2, cor, y = trn_data$y)
```



While many of these correlations are small, many very close to zero, some are as large as 0.40. Since our training data has 50 observations, we'll select the 25 predictors with the largest (absolute) correlations.

```
selected = order(abs(correlations), decreasing = TRUE)[1:25]
correlations[selected]
```

```
##      X4942       X867       X8617       X8044        X406       X4358       X7725
##  0.4005771  0.3847397  0.3809371  0.3692479 -0.3571329  0.3553777 -0.3459522
##      X1986       X3784         X77       X7010       X9354       X8450       X2355
## -0.3448612  0.3298109 -0.3252776 -0.3242813  0.3227353  0.3220087  0.3192606
##      X4381       X2486       X5947       X5767       X1227       X1464       X8223
##  0.3157441  0.3149892  0.3131235  0.3114936 -0.3105052 -0.3104528  0.3084551
```

```
##       X188      X4203      X2234      X1098
##  0.3065491  0.3039848 -0.3036512 -0.3036153
```

We subset the training and test sets to contain only the response as well as these 25 predictors.

```
trn_screen = trn_data[c(1, selected)]
tst_screen = tst_data[c(1, selected)]
```

Then we finally fit an additive logistic regression using this subset of predictors. We perform 10-fold cross-validation to obtain an estimate of the classification error.

```
add_log_mod = glm(y ~ ., data = trn_screen, family = "binomial")
boot::cv.glm(trn_screen, add_log_mod, K = 10)$delta[1]
```

```
## [1] 0.3742339
```

The 10-fold cross-validation is suggesting a classification error estimate of almost 30%.

```
add_log_pred = (predict(add_log_mod, newdata = tst_screen, type = "response") > 0.5) *
calc_misclass(predicted = add_log_pred, actual = tst_screen$y)
```

```
## [1] 0.48
```

However, if we obtain an estimate of the error using the set, we see an error rate of about 50%. No better than guessing! But since Y has no relationship with the predictors, this is actually what we would expect. This incorrect method we'll call screen-then-validate.

Now, we will correctly screen-while-validating. Essentially, instead of simply cross-validating the logistic regression, we also need to cross validate the screening process. That is, we won't simply use the same variables for each fold, we get the "best" predictors for each fold.

For methods that do not have a built-in ability to perform cross-validation, or for methods that have limited cross-validation capability, we will need to write our own code for cross-validation. (Spoiler: This is not completely true, but let's pretend it is, so we can see how to perform cross-validation from scratch.)

This essentially amounts to randomly splitting the data, then looping over the splits. The `createFolds()` function from the `caret()` package will make this much easier.

```
caret::createFolds(trn_data$y, k = 10)
```

```
## $Fold01
##  [1] 17 23 27 44 45 76 85 87 93 97
##
## $Fold02
```

```
##  [1]  6 14 15 26 37 38 55 68 69 71
##
## $Fold03
##  [1]  3  4  7 29 39 52 54 57 59 82
##
## $Fold04
##  [1] 19 21 40 46 48 56 73 78 91 96
##
## $Fold05
##  [1] 25 34 36 58 61 65 66 75 83 89
##
## $Fold06
##  [1]  2  9 10 62 74 79 80 90 92 98
##
## $Fold07
##  [1]  8 31 32 41 43 53 60 67 88 95
##
## $Fold08
##  [1] 12 18 33 35 42 49 51 64 84 94
##
## $Fold09
##  [1]  11  13  16  20  28  47  50  77  99 100
##
## $Fold10
##  [1]  1  5 22 24 30 63 70 72 81 86
```

```
# use the caret package to obtain 10 "folds"
folds = caret::createFolds(trn_data$y, k = 10)

# for each fold
# - pre-screen variables on the 9 training folds
# - fit model to these variables
# - get error on validation fold
fold_err = rep(0, length(folds))

for (i in seq_along(folds)) {

  # split for fold i
  est_fold = trn_data[-folds[[i]], ]
  val_fold = trn_data[folds[[i]], ]

  # screening for fold i
  correlations = apply(est_fold[, -1], 2, cor, y = est_fold[,1])
  selected = order(abs(correlations), decreasing = TRUE)[1:25]
  est_fold_screen = est_fold[ , c(1, selected)]
  val_fold_screen = val_fold[ , c(1, selected)]
```

```
  # error for fold i
  add_log_mod = glm(y ~ ., data = est_fold_screen, family = "binomial")
  add_log_prob = predict(add_log_mod, newdata = val_fold_screen, type = "response")
  add_log_pred = ifelse(add_log_prob > 0.5, yes = 1, no = 0)
  fold_err[i] = mean(add_log_pred != val_fold_screen$y)

}

# report all 10 validation fold errors
fold_err
```

```
##  [1] 0.4 0.9 0.6 0.4 0.6 0.3 0.7 0.5 0.6 0.6
```

```
# properly cross-validated error
# this roughly matches what we expect in the test set
mean(fold_err)
```

```
## [1] 0.56
```

- TODO: note that, even cross-validated correctly, this isn't a brilliant variable selection procedure. (it completely ignores interactions and correlations among the predictors. however, if it works, it works.) next chapters...
- TODO: calculate test error

12.5 MISC TODOS

- TODO: https://github.com/topepo/caret/issues/70
- TODO: https://stats.stackexchange.com/questions/266225/step-by-step-explanation-of-k-fold-cro
- TODO: https://weina.me/nested-cross-validation/
- rsample::nested_cv
- http://appliedpredictivemodeling.com/blog/2014/11/27/08ks7leh0zof45zpf5vqe56d1sahb0
- http://appliedpredictivemodeling.com/blog/2014/11/27/vpuig01pqbklmi72b8lcl3ij5hj2qm
- http://appliedpredictivemodeling.com/blog/2017/9/2/njdc83d01pzysvvlgik02t5qnaljnd

$$x, \boldsymbol{x}, X, \boldsymbol{X}, \mathbf{x}, \mathbf{X}, \mathbb{X}, \mathcal{X}, \mathfrak{x}, \mathfrak{X}, \mathrm{x}, \mathrm{X}$$

Chapter 13

Supervised Learning

- TODO: write an overview / review of the previous two chapters
- TODO: do two analyses?
- TODO: use caret in this chapter

Chapter 14

Regularization

14.1 STAT 432 Materials

- ISL Readings: Sections 6.1 - 6.4

```
library("tidyverse")
library("glmnet")
library("broom")
library("kableExtra")
```

14.2 Reducing Variance with Added Bias

```
gen_simple_data = function(sample_size = 25) {
  x = runif(n = sample_size)
  y = 0 + 5 * x + rnorm(n = sample_size)
  data.frame(x, y)
}
```

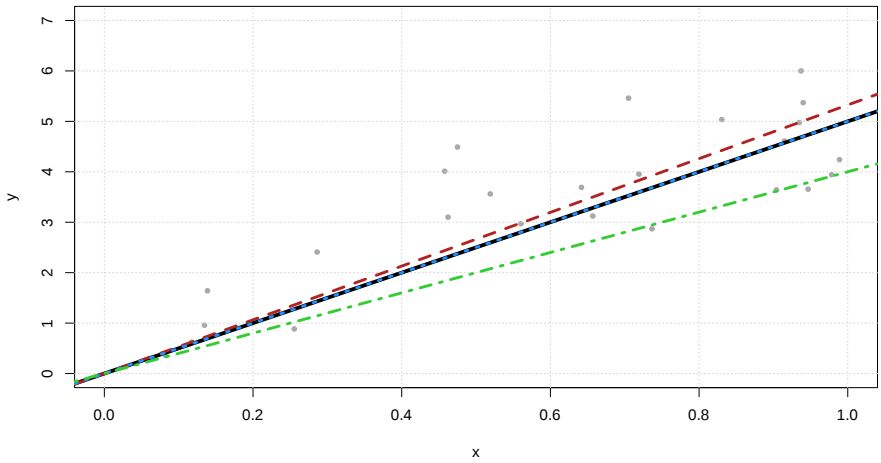
```
set.seed(42)
simple_data = gen_simple_data()
```

```
# fit least squares
beta_ls = lm(y ~ 0 + x, data = simple_data)
```

```r
# fit a biased model
# restrict beta-hat to be at most 5
beta_05 = lm(y ~ 0 + x, data = simple_data)
beta_05$coefficients = min(beta_05$coefficients, 5)

# fit a biased model
# restrict beta-hat to be at most 4
beta_04 = lm(y ~ 0 + x, data = simple_data)
beta_04$coefficients = min(beta_04$coefficients, 4)
```

```r
map_dbl(list(beta_ls, beta_05, beta_04), coef)
```

```
## [1] 5.325953 5.000000 4.000000
```




```r
# maybe write a function for each
# should switch this to make each model fit to the same data
set.seed(42)
beta_estimates = list(
  beta_ls = replicate(n = 5000, coef(lm(y ~ 0 + x, data = gen_simple_data()))["x"]),
  beta_05 = replicate(n = 5000, min(coef(lm(y ~ 0 + x, data = gen_simple_data()))["x"]
  beta_04 = replicate(n = 5000, min(coef(lm(y ~ 0 + x, data = gen_simple_data()))["x"]
)
```

Model	Bias	Variance	MSE
Least Squares	-0.003	0.13	0.130
Biased to 5	-0.134	0.04	0.058
Biased to 4	-1.000	0.00	1.001

14.3 scaling matters?

```
another_dgp = function(sample_size = 25) {
  x = runif(n = sample_size)
  y = -2 + 5 * x + rnorm(n = sample_size)
  tibble(x, y)
}
```

```
data_for_scaling = another_dgp()
predict(lm(y ~ x, data = data_for_scaling))
```

```
##          1          2          3          4          5          6          7
## -1.3821582 -1.9123198  2.3840155 -1.9167476  2.6436459 -0.0673255 -0.2236885
##          8          9         10         11         12         13         14
##  2.4778373  1.1021074  1.6899061  0.2489363 -0.7390997  3.4760505  0.4906860
##         15         16         17         18         19         20         21
## -0.9683105  0.4202554  1.7825769 -0.2720839  0.9580110  1.1225404  2.9943423
##         22         23         24         25
##  3.7554539  1.2570305 -0.9414314  2.3061219
```

```
coef(lm(y ~ x, data = data_for_scaling))
```

```
## (Intercept)           x
##   -2.229674    6.287082
```

```
data_for_scaling$x = scale(data_for_scaling$x)
predict(lm(y ~ x, data = data_for_scaling))
```

```
##          1          2          3          4          5          6          7
## -1.3821582 -1.9123198  2.3840155 -1.9167476  2.6436459 -0.0673255 -0.2236885
##          8          9         10         11         12         13         14
##  2.4778373  1.1021074  1.6899061  0.2489363 -0.7390997  3.4760505  0.4906860
##         15         16         17         18         19         20         21
## -0.9683105  0.4202554  1.7825769 -0.2720839  0.9580110  1.1225404  2.9943423
##         22         23         24         25
##  3.7554539  1.2570305 -0.9414314  2.3061219
```

```
coef(lm(y ~ x, data = data_for_scaling))
```

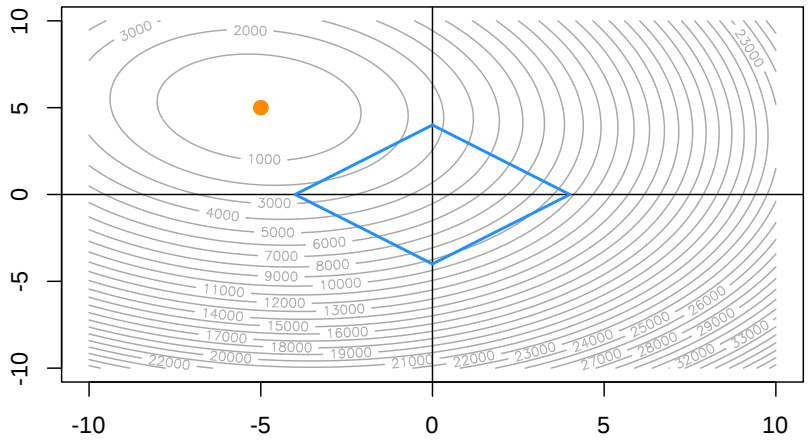
```
## (Intercept)           x
##   0.8274541   1.6506452
```

14.4 Constraints in Two Dimensions

```
gen_linear_data = function() {
  x1 = rnorm(100)
  x2 = rnorm(100)
  y = 0 + -5 * x1 + 5 * x2 + rnorm(100)
  tibble(x1 = x1, x2 = x2, y = y)
}
```

```
data = gen_linear_data()
beta = expand.grid(beta_1 = seq(-10, 10, 0.1),
                   beta_2 = seq(-10, 10, 0.1))
beta_error = rep(0, dim(beta)[1])
for (i in 1:dim(beta)[1]){
  beta_error[i] = with(data, sum((y - (beta$beta_1[i] * x1 + beta$beta_2[i] * x2)) ^ 2
}
```

```
# TODO: make this into a function
# TODO: add ridge constraint
contour(x = seq(-10, 10, 0.1),
        y = seq(-10, 10, 0.1),
        z = matrix(beta_error,
                   nrow = length(seq(-10, 10, 0.1)),
                   ncol = length(seq(-10, 10, 0.1))),
        nlevels = 50,
        col = "darkgrey"
)

abline(h = 0)
abline(v = 0)
a = 4
segments(0, a, a, 0, col = "dodgerblue", lwd = 2)
segments(0, -a, a, 0, col = "dodgerblue", lwd = 2)
segments(-a, 0, 0, a, col = "dodgerblue", lwd = 2)
segments(-a, 0, 0, -a, col = "dodgerblue", lwd = 2)
points(beta[which.min(beta_error), ], col = "darkorange", pch = 20, cex = 2)
```



14.5 High Dimensional Data

```
gen_wide_data = function(sample_size = 100, sig_betas = 5, p = 200) {

  if (p <= sample_size) {
    warning("You're not generating wide data, despite the name of the function.")
  }

  if (sig_betas > p) {
    stop("Cannot have more significant variables than variables!")
  }

  x = map_dfc(1:p, ~ rnorm(n = sample_size))
  x = x %>% rename_all( ~ str_replace(., "V", "x"))
  sig_x = x[, 1:sig_betas]
  beta = rep(3, times = sig_betas)
  y = as.matrix(sig_x) %*% beta + rnorm(n = sample_size)
  bind_cols(y = y, x)
}
```

```
some_wide_data = gen_wide_data()
```

```
some_wide_data
```

```
## # A tibble: 100 x 201
##      y[,1]     x1     x2      x3       x4      x5      x6     x7     x8      x9
##      <dbl>  <dbl>  <dbl>   <dbl>    <dbl>   <dbl>   <dbl>  <dbl>  <dbl>  <dbl>
##  1  5.00    1.75  -0.449  1.09   -0.981    0.217  -0.390   1.13   1.83  -0.989
##  2 -7.81   -0.792 -0.659 -0.823  -0.275    0.281   0.931   0.680  0.354  0.628
##  3  4.29    0.648  0.533 -0.415   1.19    -0.451   0.914   0.953  0.527  0.529
##  4  7.22    1.06   0.913  0.0341  0.793   -0.522   0.680  -0.124  1.74   0.955
##  5 -2.09   -0.603 -0.594 -0.141   0.897    0.162  -0.0729 -0.550  2.81   1.71
##  6 -6.24    0.461 -0.996 -1.25   -0.386   -0.175  -0.153  -0.914  0.230  0.624
##  7 10.2     1.45   1.60   0.426   0.0269   0.318   0.0255 -2.15   0.669 -0.178
##  8  2.42    0.576  0.757  0.537  -0.828   -0.348  -0.640   2.25   0.942  1.10
##  9 -0.0202 -0.287  0.276 -0.550   0.537   -0.0493  1.42   -0.558 -0.283  0.489
## 10  1.49    0.563 -0.264  0.0107  0.00393 -0.367   1.75   -0.893  1.75   1.08
## # ... with 90 more rows, and 191 more variables: x10 <dbl>, x11 <dbl>,
## #   x12 <dbl>, x13 <dbl>, x14 <dbl>, x15 <dbl>, x16 <dbl>, x17 <dbl>,
## #   x18 <dbl>, x19 <dbl>, x20 <dbl>, x21 <dbl>, x22 <dbl>, x23 <dbl>,
## #   x24 <dbl>, x25 <dbl>, x26 <dbl>, x27 <dbl>, x28 <dbl>, x29 <dbl>,
## #   x30 <dbl>, x31 <dbl>, x32 <dbl>, x33 <dbl>, x34 <dbl>, x35 <dbl>,
## #   x36 <dbl>, x37 <dbl>, x38 <dbl>, x39 <dbl>, x40 <dbl>, x41 <dbl>,
## #   x42 <dbl>, x43 <dbl>, x44 <dbl>, x45 <dbl>, x46 <dbl>, x47 <dbl>,
## #   x48 <dbl>, x49 <dbl>, x50 <dbl>, x51 <dbl>, x52 <dbl>, x53 <dbl>,
## #   x54 <dbl>, x55 <dbl>, x56 <dbl>, x57 <dbl>, x58 <dbl>, x59 <dbl>,
## #   x60 <dbl>, x61 <dbl>, x62 <dbl>, x63 <dbl>, x64 <dbl>, x65 <dbl>,
## #   x66 <dbl>, x67 <dbl>, x68 <dbl>, x69 <dbl>, x70 <dbl>, x71 <dbl>,
## #   x72 <dbl>, x73 <dbl>, x74 <dbl>, x75 <dbl>, x76 <dbl>, x77 <dbl>,
## #   x78 <dbl>, x79 <dbl>, x80 <dbl>, x81 <dbl>, x82 <dbl>, x83 <dbl>,
## #   x84 <dbl>, x85 <dbl>, x86 <dbl>, x87 <dbl>, x88 <dbl>, x89 <dbl>,
## #   x90 <dbl>, x91 <dbl>, x92 <dbl>, x93 <dbl>, x94 <dbl>, x95 <dbl>,
## #   x96 <dbl>, x97 <dbl>, x98 <dbl>, x99 <dbl>, x100 <dbl>, x101 <dbl>,
## #   x102 <dbl>, x103 <dbl>, x104 <dbl>, x105 <dbl>, x106 <dbl>, x107 <dbl>,
## #   x108 <dbl>, x109 <dbl>, ...
```

14.6 Ridge Regression

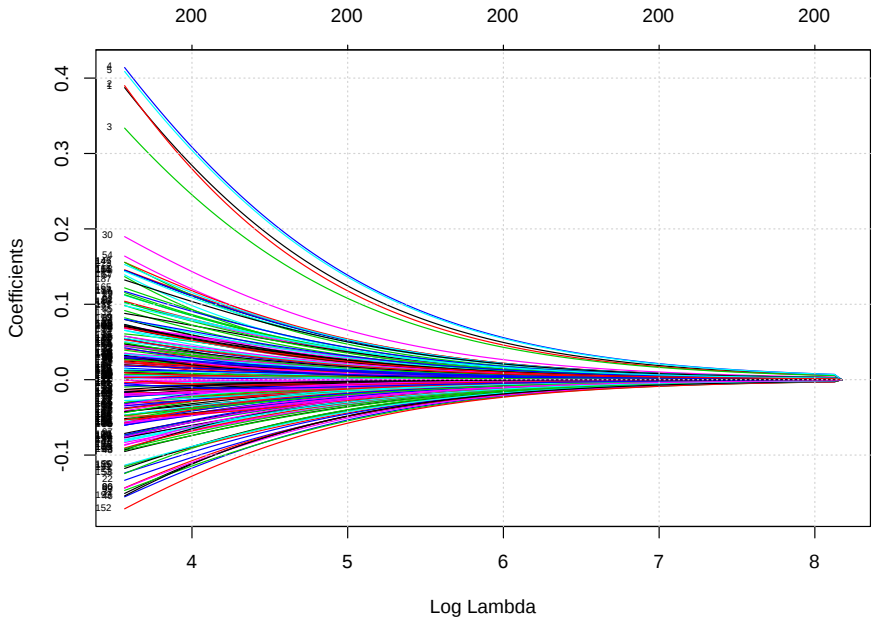
$$\sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij} \right)^2 + \lambda \sum_{j=1}^{p} \beta_j^2.$$

```
set.seed(42)
data_for_ridge = gen_wide_data(sample_size = 100, sig_betas = 5, p = 200)
```

```
x_ridge = data_for_ridge %>% select(-y) %>% as.matrix()
y_ridge = data_for_ridge %>% pull(y)
```

```
mod_ridge = glmnet(x = x_ridge, y = y_ridge, alpha = 0)
```

```
plot(mod_ridge, xvar = "lambda", label = TRUE)
grid()
```




```
as_tibble(predict(mod_ridge, x_ridge[1:5, ]))
```

```
## # A tibble: 5 x 100
##       s0     s1     s2     s3     s4     s5     s6     s7     s8     s9    s10
##    <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
## 1 -0.407 -0.357 -0.354 -0.352 -0.349 -0.346 -0.344 -0.341 -0.337 -0.334 -0.331
## 2 -0.407 -0.400 -0.400 -0.400 -0.399 -0.399 -0.398 -0.398 -0.398 -0.397 -0.397
## 3 -0.407 -0.361 -0.359 -0.356 -0.354 -0.352 -0.349 -0.346 -0.343 -0.340 -0.337
## 4 -0.407 -0.322 -0.318 -0.314 -0.310 -0.305 -0.301 -0.295 -0.290 -0.285 -0.279
## 5 -0.407 -0.441 -0.442 -0.444 -0.446 -0.447 -0.449 -0.451 -0.453 -0.456 -0.458
## # ... with 89 more variables: s11 <dbl>, s12 <dbl>, s13 <dbl>, s14 <dbl>,
## #   s15 <dbl>, s16 <dbl>, s17 <dbl>, s18 <dbl>, s19 <dbl>, s20 <dbl>,
## #   s21 <dbl>, s22 <dbl>, s23 <dbl>, s24 <dbl>, s25 <dbl>, s26 <dbl>,
## #   s27 <dbl>, s28 <dbl>, s29 <dbl>, s30 <dbl>, s31 <dbl>, s32 <dbl>,
## #   s33 <dbl>, s34 <dbl>, s35 <dbl>, s36 <dbl>, s37 <dbl>, s38 <dbl>,
## #   s39 <dbl>, s40 <dbl>, s41 <dbl>, s42 <dbl>, s43 <dbl>, s44 <dbl>,
```

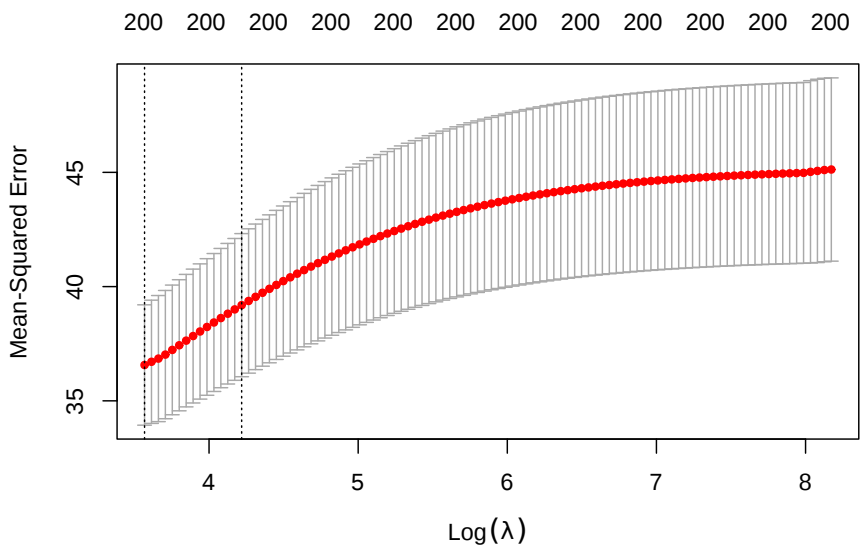
```
## #   s45 <dbl>, s46 <dbl>, s47 <dbl>, s48 <dbl>, s49 <dbl>, s50 <dbl>,
## #   s51 <dbl>, s52 <dbl>, s53 <dbl>, s54 <dbl>, s55 <dbl>, s56 <dbl>,
## #   s57 <dbl>, s58 <dbl>, s59 <dbl>, s60 <dbl>, s61 <dbl>, s62 <dbl>,
## #   s63 <dbl>, s64 <dbl>, s65 <dbl>, s66 <dbl>, s67 <dbl>, s68 <dbl>,
## #   s69 <dbl>, s70 <dbl>, s71 <dbl>, s72 <dbl>, s73 <dbl>, s74 <dbl>,
## #   s75 <dbl>, s76 <dbl>, s77 <dbl>, s78 <dbl>, s79 <dbl>, s80 <dbl>,
## #   s81 <dbl>, s82 <dbl>, s83 <dbl>, s84 <dbl>, s85 <dbl>, s86 <dbl>,
## #   s87 <dbl>, s88 <dbl>, s89 <dbl>, s90 <dbl>, s91 <dbl>, s92 <dbl>,
## #   s93 <dbl>, s94 <dbl>, s95 <dbl>, s96 <dbl>, s97 <dbl>, s98 <dbl>, s99 <dbl>
```

```
set.seed(42)
mod_ridge = cv.glmnet(x = x_ridge, y = y_ridge, alpha = 0, nfolds = 5)
```

```
glance(mod_ridge)
```

```
## # A tibble: 1 x 2
##   lambda.min lambda.1se
##        <dbl>      <dbl>
## 1       35.4       67.9
```

```
plot(mod_ridge)
```




```
tidy(mod_ridge)
```

```
## # A tibble: 100 x 6
##    lambda estimate std.error conf.low conf.high nzero
##     <dbl>    <dbl>     <dbl>    <dbl>     <dbl> <int>
```

```
##  1  3543.     45.1      4.01     41.1      49.1   200
##  2  3382.     45.1      4.02     41.1      49.1   200
##  3  3228.     45.1      4.01     41.1      49.1   200
##  4  3081.     45.0      3.99     41.0      49.0   200
##  5  2941.     45.0      3.96     41.0      48.9   200
##  6  2808.     45.0      3.96     41.0      48.9   200
##  7  2680.     45.0      3.95     41.0      48.9   200
##  8  2558.     44.9      3.95     41.0      48.9   200
##  9  2442.     44.9      3.95     41.0      48.9   200
## 10  2331.     44.9      3.95     41.0      48.9   200
## # ... with 90 more rows
```

14.7 Lasso

14.8 boston is boring

```
bstn = MASS::Boston

bstn$chas = factor(bstn$chas)
bstn$rad = factor(bstn$rad)

levels(bstn$chas)
```

```
## [1] "0" "1"
```

```
levels(bstn$rad)
```

```
## [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "24"
```

```
lm(medv ~ ., data = bstn)
```

```
##
## Call:
## lm(formula = medv ~ ., data = bstn)
##
## Coefficients:
## (Intercept)         crim           zn        indus        chas1          nox
##   35.259615    -0.108821     0.054896     0.023760     2.524163   -17.573132
##          rm          age          dis         rad2         rad3         rad4
##    3.665491     0.000461    -1.554546     1.488905     4.681253     2.576234
##        rad5         rad6         rad7         rad8        rad24          tax
##    2.918493     1.185839     4.878992     4.839836     7.461674    -0.008748
##     ptratio        black        lstat
##   -0.972419     0.009394    -0.529226
```

```
head(as_tibble(model.matrix(lm(medv ~ ., data = bstn))))
```

```
## # A tibble: 6 x 21
##   `(Intercept)`    crim    zn indus chas1   nox    rm   age   dis  rad2  rad3
##           <dbl>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1             1 0.00632    18  2.31     0 0.538  6.58  65.2  4.09     0     0
## 2             1 0.0273      0  7.07     0 0.469  6.42  78.9  4.97     1     0
## 3             1 0.0273      0  7.07     0 0.469  7.18  61.1  4.97     1     0
## 4             1 0.0324      0  2.18     0 0.458  7.00  45.8  6.06     0     1
## 5             1 0.0690      0  2.18     0 0.458  7.15  54.2  6.06     0     1
## 6             1 0.0298      0  2.18     0 0.458  6.43  58.7  6.06     0     1
## # ... with 10 more variables: rad4 <dbl>, rad5 <dbl>, rad6 <dbl>, rad7 <dbl>,
## #   rad8 <dbl>, rad24 <dbl>, tax <dbl>, ptratio <dbl>, black <dbl>, lstat <dbl>
```

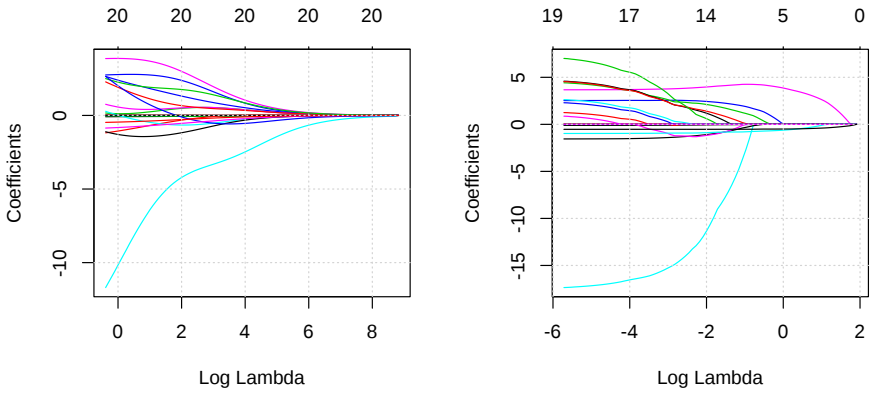
```
bstn_x = model.matrix(lm(medv ~ ., data = bstn))
bstn_y = bstn$medv
```

```
coef(lm.fit(x = bstn_x, y = bstn_y))
```

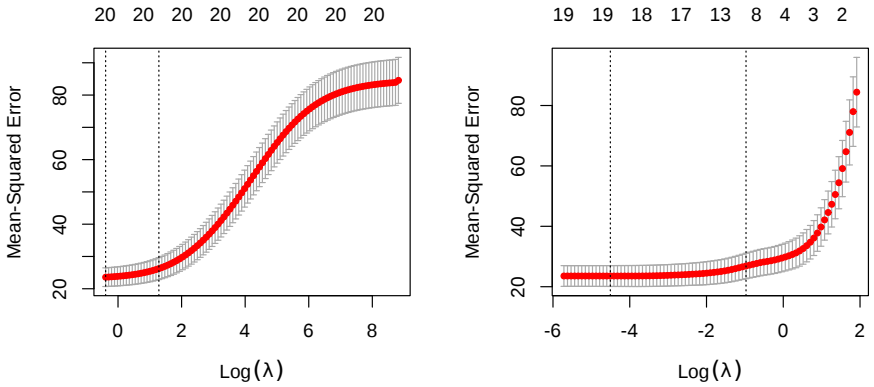
```
##   (Intercept)          crim            zn         indus         chas1
##  3.525962e+01 -1.088210e-01  5.489638e-02  2.376030e-02  2.524163e+00
##           nox            rm           age           dis          rad2
## -1.757313e+01  3.665491e+00  4.610055e-04 -1.554546e+00  1.488905e+00
##          rad3          rad4          rad5          rad6          rad7
##  4.681253e+00  2.576234e+00  2.918493e+00  1.185839e+00  4.878992e+00
##          rad8         rad24           tax       ptratio         black
##  4.839836e+00  7.461674e+00 -8.748175e-03 -9.724194e-01  9.393803e-03
##         lstat
## -5.292258e-01
```

```
bstn_x = model.matrix(lm(medv ~ ., data = bstn))[, -1]
bstn_y = bstn$medv
```

```
par(mfrow = c(1, 2))
plot(glmnet(x = bstn_x, y = bstn_y, alpha = 0), xvar = "lambda")
grid()
plot(glmnet(x = bstn_x, y = bstn_y, alpha = 1), xvar = "lambda")
grid()
```




```
par(mfrow = c(1, 2))
plot(cv.glmnet(x = bstn_x, y = bstn_y, alpha = 0))
plot(cv.glmnet(x = bstn_x, y = bstn_y, alpha = 1))
```




```
bstn_ridge = cv.glmnet(x = bstn_x, y = bstn_y, alpha = 0)
bstn_lasso = cv.glmnet(x = bstn_x, y = bstn_y, alpha = 1)
```

```
library("broom")
```

```
tidy(bstn_lasso)
```

```
## # A tibble: 83 x 6
##    lambda estimate std.error conf.low conf.high nzero
##     <dbl>    <dbl>     <dbl>    <dbl>     <dbl> <int>
##  1   6.78     84.3      8.10     76.2      92.4     0
##  2   6.18     77.2      7.94     69.2      85.1     1
```

```
##  3   5.63     70.4      7.51     62.9      77.9     2
##  4   5.13     64.0      7.06     57.0      71.1     2
##  5   4.67     58.5      6.59     51.9      65.0     2
##  6   4.26     53.8      6.17     47.6      60.0     2
##  7   3.88     49.9      5.82     44.1      55.8     2
##  8   3.53     46.7      5.53     41.2      52.3     2
##  9   3.22     44.1      5.28     38.8      49.3     2
## 10   2.93     41.6      5.08     36.5      46.7     3
## # ... with 73 more rows
```

```
glance(bstn_lasso)
```

```
## # A tibble: 1 x 2
##   lambda.min lambda.1se
##        <dbl>      <dbl>
## 1    0.00694      0.315
```

```
# TODO: pull out rows of tidy with the values from glance
```

```
predict(bstn_lasso, newx = bstn_x[1:10,], type = "link")
```

```
##           1
## 1  30.39658
## 2  25.38272
## 3  31.23894
## 4  31.17211
## 5  30.57321
## 6  27.58938
## 7  23.54446
## 8  20.54428
## 9  12.51911
## 10 20.51929
```

```
predict(bstn_lasso, newx = bstn_x[1:10,], type = "response")
```

```
##           1
## 1  30.39658
## 2  25.38272
## 3  31.23894
## 4  31.17211
## 5  30.57321
## 6  27.58938
## 7  23.54446
## 8  20.54428
## 9  12.51911
## 10 20.51929
```

```
predict(bstn_lasso, type = "coefficients", s=c("lambda.1se","lambda.min"))
```

```
## 21 x 1 sparse Matrix of class "dgCMatrix"
##                          1
## (Intercept) 19.386070871
## crim        -0.027179560
## zn           0.004205338
## indus        .
## chas1        2.033274012
## nox         -4.744669654
## rm           4.223773143
## age          .
## dis         -0.485529479
## rad2         .
## rad3         1.353021994
## rad4         .
## rad5         .
## rad6        -0.286318856
## rad7         .
## rad8         0.336943107
## rad24        .
## tax          .
## ptratio     -0.810303776
## black        0.006775041
## lstat       -0.519927242
```

```
predict(bstn_lasso, type = "nonzero")
```

```
##    X1
## 1   1
## 2   2
## 3   4
## 4   5
## 5   6
## 6   8
## 7  10
## 8  13
## 9  15
## 10 18
## 11 19
## 12 20
```

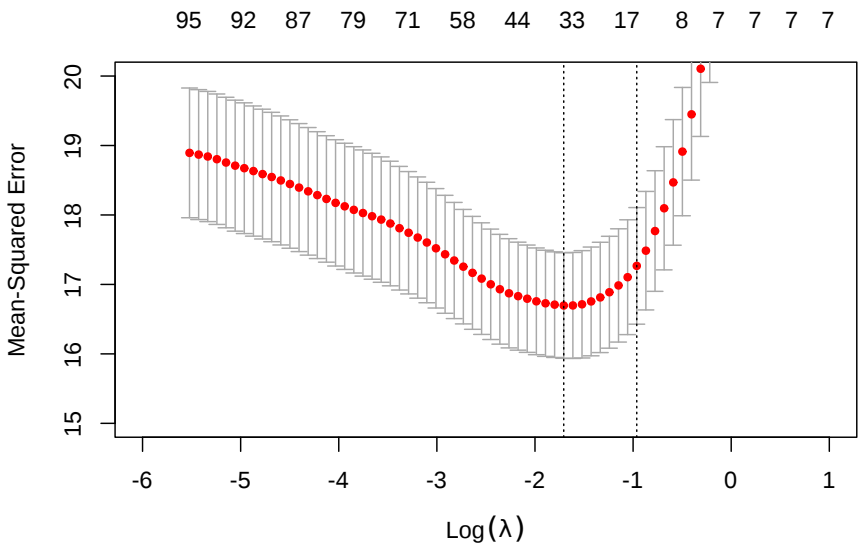
14.9 some more simulation

```
# diag(100)
```

```
p = 100
A = matrix(runif(p ^ 2) * 2 - 1, ncol = p)
Sigma = t(A) %*% A
sample_size = 500
X = MASS::mvrnorm(n = sample_size, mu = rep(0, p), Sigma = Sigma)
beta = ifelse(sample(c(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1), size = p, replace = TRU
y = X %*% beta + rnorm(n = sample_size, sd = 4)
fit = glmnet::cv.glmnet(x = X, y = y, alpha = 1)
sqrt(min(fit$cvm))
```

```
## [1] 4.086107
```

```
plot(fit, xlim = c(-6, 1), ylim = c(15, 20))
```




```
# type.measure = "class"
```

- TODO: Least Absolute Shrinkage and Selection Operator
- TODO: https://statisticaloddsandends.wordpress.com/2018/11/15/a-deep-dive-into-glmnet-standardize/
- TODO: https://www.jaredlander.com/2018/02/using-coefplot-with-glmnet/
- TODO: statistical learning with sparsity book

Chapter 15

Ensembles

15.1 STAT 432 Materials

- **Slides** | Ensemble Methods
- ISL Readings: Sections 8.1 - 8.2

```
library("tibble")
library("rpart")
library("rpart.plot")
library("caret")
library("purrr")
library("randomForest")
library("gbm")
library("xgboost")
library("knitr")
library("kableExtra")
```

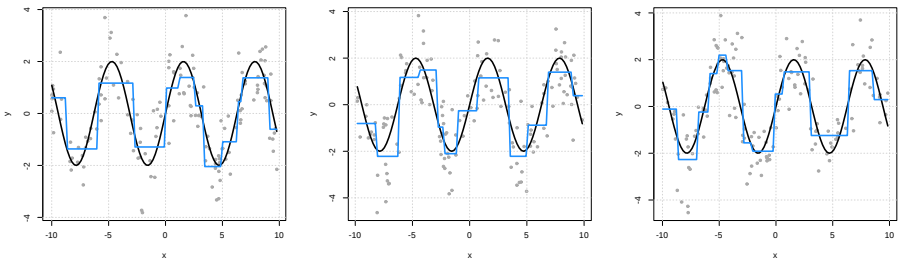
15.2 Bagging

```
sin_dgp = function(sample_size = 150) {
  x = runif(n = sample_size, min = -10, max = 10)
  y = 2 * sin(x) + rnorm(n = sample_size)
```

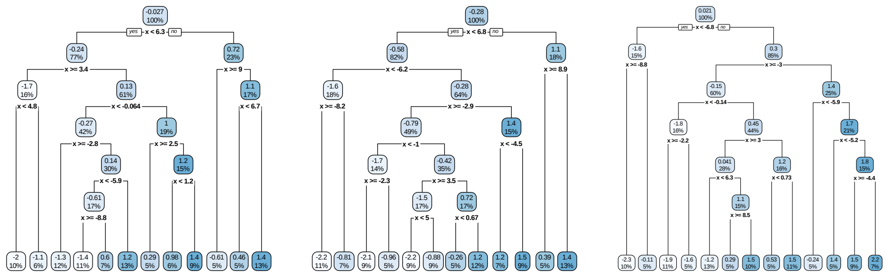
```
  tibble(x = x, y = y)
}
```

```
set.seed(42)
```

```
par(mfrow = c(1, 3))

some_data = sin_dgp()
plot(some_data, pch = 20, col = "darkgrey")
grid()
curve(2 * sin(x), add = TRUE, col = "black", lwd = 2)
fit_1 = rpart(y ~ x, data = some_data, cp = 0)
curve(predict(fit_1, tibble(x = x)), add = TRUE, lwd = 2, col = "dodgerblue")

some_data = sin_dgp()
plot(some_data, pch = 20, col = "darkgrey")
grid()
curve(2 * sin(x), add = TRUE, col = "black", lwd = 2)
fit_2 = rpart(y ~ x, data = some_data, cp = 0)
curve(predict(fit_2, tibble(x = x)), add = TRUE, lwd = 2, col = "dodgerblue")

some_data = sin_dgp()
plot(some_data, pch = 20, col = "darkgrey")
grid()
curve(2 * sin(x), add = TRUE, col = "black", lwd = 2)
fit_3 = rpart(y ~ x, data = some_data, cp = 0)
curve(predict(fit_3, tibble(x = x)), add = TRUE, lwd = 2, col = "dodgerblue")
```



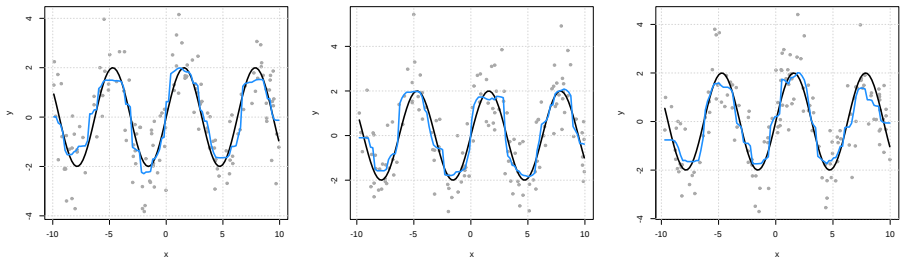
```
par(mfrow = c(1, 3))

rpart.plot(fit_1)
rpart.plot(fit_2)
rpart.plot(fit_3)
```



```
bag_pred = function(x) {
  apply(t(map_df(boot_reps, predict, data.frame(x = x))), 2, mean)
}

set.seed(42)
boot_idx = caret::createResample(y = some_data$y, times = 100)
boot_reps = map(boot_idx, ~ rpart(y ~ x, data = some_data[.x, ], cp = 0))
bag_pred(x = c(-1, 0 , 1))
```

```
## [1] -1.87295394  0.06556445  1.18010317
```

```
par(mfrow = c(1, 3))

some_data = sin_dgp()
plot(some_data, pch = 20, col = "darkgrey")
grid()
curve(2 * sin(x), add = TRUE, col = "black", lwd = 2)
boot_idx = caret::createResample(y = some_data$y, times = 100)
boot_reps = map(boot_idx, ~ rpart(y ~ x, data = some_data[.x, ], cp = 0))
curve(bag_pred(x = x), add = TRUE, lwd = 2, col = "dodgerblue")

some_data = sin_dgp()
plot(some_data, pch = 20, col = "darkgrey")
grid()
curve(2 * sin(x), add = TRUE, col = "black", lwd = 2)
boot_idx = caret::createResample(y = some_data$y, times = 100)
boot_reps = map(boot_idx, ~ rpart(y ~ x, data = some_data[.x, ], cp = 0))
curve(bag_pred(x = x), add = TRUE, lwd = 2, col = "dodgerblue")

some_data = sin_dgp()
plot(some_data, pch = 20, col = "darkgrey")
grid()
curve(2 * sin(x), add = TRUE, col = "black", lwd = 2)
boot_idx = caret::createResample(y = some_data$y, times = 100)
boot_reps = map(boot_idx, ~ rpart(y ~ x, data = some_data[.x, ], cp = 0))
curve(bag_pred(x = x), add = TRUE, lwd = 2, col = "dodgerblue")
```



15.2.1 Simultation Study

```
new_obs = tibble(x = 0, y = (2 * sin(0)))
```

```
sim_bagging_vs_single = function() {
  some_data = sin_dgp()

  single = predict(rpart(y ~ x, data = some_data, cp = 0), new_obs)

  boot_idx = caret::createResample(y = some_data$y, times = 100)
  boot_reps = map(boot_idx, ~ rpart(y ~ x, data = some_data[.x, ], cp = 0))
  bagged = mean(map_dbl(boot_reps, predict, new_obs))
  c(single = single, bagged = bagged)
}

set.seed(42)
sim_results = replicate(n = 250, sim_bagging_vs_single())
apply(sim_results, 1, mean)
```

```
##    single.1      bagged
## -0.02487810  0.03302126
```

```
apply(sim_results, 1, var)
```

```
##  single.1    bagged
## 1.0147130 0.3356579
```

15.3 Random Forest

```
set.seed(42)
two_class_data = as_tibble(caret::twoClassSim(n = 1250, noiseVars = 20))
two_class_data
```

```
## # A tibble: 1,250 x 36
##    TwoFactor1 TwoFactor2 Linear01 Linear02 Linear03 Linear04 Linear05 Linear06
##         <dbl>      <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>
##  1      2.68       0.843  0.617      0.761   0.0712   -0.368    0.294    2.44
##  2     -0.496     -0.955 -0.00454   -0.172   0.970    -1.28    -0.404    2.36
##  3     -0.577      1.51  -0.0913    -0.265   0.310    -1.98    -2.18     1.49
##  4      1.06       0.567  0.400     -0.424  -0.140    -1.24     0.198    0.931
##  5      1.21      -0.171  0.589      0.689  -0.326    -0.541    1.92    -1.40
##  6     -0.161     -0.112 -0.0169     1.05   -0.119    -0.219   -0.928    1.32
##  7      1.20       2.68   0.419     -2.19    0.894    -1.30    -0.329   -0.319
##  8      0.812     -1.06   0.801     -0.503   0.211     1.45     0.952    0.135
##  9      2.74       2.44   0.680      0.880  -0.489    -0.197    1.64    -2.64
## 10     -0.357      0.196  1.31       1.55   -0.220     0.891   -0.764    1.24
## # ... with 1,240 more rows, and 28 more variables: Linear07 <dbl>,
## #   Linear08 <dbl>, Linear09 <dbl>, Linear10 <dbl>, Nonlinear1 <dbl>,
## #   Nonlinear2 <dbl>, Nonlinear3 <dbl>, Noise01 <dbl>, Noise02 <dbl>,
## #   Noise03 <dbl>, Noise04 <dbl>, Noise05 <dbl>, Noise06 <dbl>, Noise07 <dbl>,
## #   Noise08 <dbl>, Noise09 <dbl>, Noise10 <dbl>, Noise11 <dbl>, Noise12 <dbl>,
## #   Noise13 <dbl>, Noise14 <dbl>, Noise15 <dbl>, Noise16 <dbl>, Noise17 <dbl>,
## #   Noise18 <dbl>, Noise19 <dbl>, Noise20 <dbl>, Class <fct>
```

```
fit = randomForest(Class ~ ., data = two_class_data)
fit
```

```
##
## Call:
##  randomForest(formula = Class ~ ., data = two_class_data)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 5
##
##         OOB estimate of  error rate: 18.24%
## Confusion matrix:
##        Class1 Class2 class.error
## Class1    562     96   0.1458967
## Class2    132    460   0.2229730
```

```
all.equal(predict(fit), predict(fit, two_class_data))
```

```
## [1] "228 string mismatches"
```

```
tibble(
  "Training Observation" = 1:10,
  "OOB Predictions" = head(predict(fit), n = 10),
  "Full Forest Predictions" = head(predict(fit, two_class_data), n = 10)
) %>%
  kable() %>%
  kable_styling("striped", full_width = FALSE)
```

Training Observation	OOB Predictions	Full Forest Predictions
1	Class1	Class1
2	Class1	Class1
3	Class2	Class2
4	Class1	Class1
5	Class2	Class1
6	Class1	Class1
7	Class2	Class2
8	Class1	Class1
9	Class2	Class2
10	Class1	Class1

```
predict(fit, two_class_data, type = "prob")[2, ]
```

```
## Class1 Class2
##  0.822  0.178
```

```
predict(fit, two_class_data, predict.all = TRUE)$individual[2, ]
```

```
##   [1] "Class2" "Class1" "Class1" "Class1" "Class1" "Class1" "Class1" "Class1"
##   [9] "Class1" "Class1" "Class1" "Class2" "Class1" "Class1" "Class1" "Class1"
##  [17] "Class1" "Class1" "Class1" "Class1" "Class2" "Class1" "Class2" "Class1"
##  [25] "Class1" "Class1" "Class1" "Class1" "Class1" "Class1" "Class1" "Class1"
##  [33] "Class2" "Class1" "Class1" "Class1" "Class1" "Class1" "Class1" "Class1"
##  [41] "Class1" "Class2" "Class2" "Class1" "Class1" "Class1" "Class1" "Class1"
##  [49] "Class1" "Class1" "Class2" "Class2" "Class2" "Class1" "Class1" "Class1"
##  [57] "Class1" "Class1" "Class2" "Class2" "Class1" "Class1" "Class1" "Class1"
##  [65] "Class1" "Class1" "Class1" "Class1" "Class1" "Class1" "Class1" "Class1"
##  [73] "Class1" "Class2" "Class1" "Class1" "Class1" "Class1" "Class1" "Class1"
##  [81] "Class1" "Class1" "Class1" "Class1" "Class1" "Class1" "Class2" "Class1"
##  [89] "Class1" "Class1" "Class1" "Class1" "Class1" "Class1" "Class2" "Class1"
##  [97] "Class1" "Class1" "Class1" "Class1" "Class2" "Class1" "Class1" "Class1"
## [105] "Class1" "Class1" "Class2" "Class1" "Class2" "Class1" "Class2" "Class1"
## [113] "Class1" "Class1" "Class1" "Class1" "Class1" "Class1" "Class1" "Class2"
## [121] "Class1" "Class1" "Class1" "Class2" "Class1" "Class1" "Class1" "Class1"
## [129] "Class2" "Class1" "Class1" "Class1" "Class2" "Class1" "Class1" "Class1"
## [137] "Class1" "Class2" "Class2" "Class1" "Class1" "Class1" "Class1" "Class1"
## [145] "Class1" "Class2" "Class1" "Class1" "Class1" "Class1" "Class1" "Class1"
## [153] "Class1" "Class1" "Class1" "Class1" "Class1" "Class1" "Class1" "Class1"
## [161] "Class1" "Class1" "Class1" "Class1" "Class1" "Class1" "Class1" "Class1"
## [169] "Class1" "Class2" "Class1" "Class1" "Class1" "Class2" "Class1" "Class1"
## [177] "Class1" "Class1" "Class2" "Class2" "Class1" "Class1" "Class1" "Class2"
## [185] "Class1" "Class1" "Class2" "Class1" "Class1" "Class2" "Class1" "Class2"
## [193] "Class1" "Class1" "Class1" "Class1" "Class1" "Class1" "Class1" "Class1"
```

```
## [201] "Class1" "Class1" "Class1" "Class1" "Class1" "Class2" "Class1" "Class1"
## [209] "Class1" "Class1" "Class1" "Class1" "Class1" "Class1" "Class1" "Class1"
## [217] "Class1" "Class1" "Class2" "Class1" "Class1" "Class2" "Class2" "Class2"
## [225] "Class1" "Class1" "Class1" "Class1" "Class2" "Class1" "Class1" "Class1"
## [233] "Class1" "Class1" "Class1" "Class1" "Class2" "Class2" "Class1" "Class1"
## [241] "Class1" "Class1" "Class1" "Class1" "Class1" "Class1" "Class1" "Class1"
## [249] "Class1" "Class1" "Class1" "Class1" "Class1" "Class1" "Class2" "Class1"
## [257] "Class1" "Class1" "Class2" "Class1" "Class1" "Class1" "Class1" "Class1"
## [265] "Class1" "Class1" "Class2" "Class1" "Class1" "Class1" "Class1" "Class1"
## [273] "Class2" "Class1" "Class2" "Class1" "Class1" "Class1" "Class1" "Class1"
## [281] "Class1" "Class1" "Class1" "Class1" "Class2" "Class1" "Class1" "Class1"
## [289] "Class2" "Class1" "Class1" "Class1" "Class1" "Class1" "Class1" "Class1"
## [297] "Class1" "Class1" "Class1" "Class1" "Class2" "Class1" "Class1" "Class1"
## [305] "Class1" "Class1" "Class1" "Class1" "Class1" "Class1" "Class1" "Class1"
## [313] "Class1" "Class2" "Class2" "Class1" "Class1" "Class1" "Class1" "Class1"
## [321] "Class1" "Class1" "Class1" "Class1" "Class1" "Class1" "Class2" "Class2"
## [329] "Class1" "Class1" "Class1" "Class1" "Class1" "Class2" "Class1" "Class1"
## [337] "Class1" "Class2" "Class1" "Class1" "Class1" "Class1" "Class1" "Class1"
## [345] "Class1" "Class1" "Class1" "Class1" "Class1" "Class1" "Class1" "Class2"
## [353] "Class1" "Class1" "Class1" "Class1" "Class1" "Class2" "Class1" "Class1"
## [361] "Class1" "Class1" "Class2" "Class1" "Class1" "Class1" "Class1" "Class2"
## [369] "Class1" "Class1" "Class1" "Class2" "Class1" "Class1" "Class2" "Class2"
## [377] "Class1" "Class2" "Class2" "Class1" "Class1" "Class1" "Class1" "Class2"
## [385] "Class2" "Class1" "Class1" "Class2" "Class2" "Class2" "Class1" "Class2"
## [393] "Class1" "Class1" "Class2" "Class1" "Class1" "Class1" "Class1" "Class2"
## [401] "Class2" "Class1" "Class1" "Class2" "Class1" "Class2" "Class1" "Class1"
## [409] "Class1" "Class1" "Class1" "Class2" "Class1" "Class1" "Class1" "Class2"
## [417] "Class1" "Class1" "Class1" "Class1" "Class1" "Class1" "Class1" "Class1"
## [425] "Class1" "Class1" "Class1" "Class1" "Class1" "Class1" "Class1" "Class1"
## [433] "Class2" "Class1" "Class1" "Class2" "Class1" "Class2" "Class1" "Class1"
## [441] "Class1" "Class1" "Class1" "Class1" "Class1" "Class1" "Class1" "Class1"
## [449] "Class1" "Class1" "Class1" "Class1" "Class2" "Class2" "Class1" "Class1"
## [457] "Class1" "Class1" "Class1" "Class1" "Class1" "Class1" "Class1" "Class1"
## [465] "Class1" "Class1" "Class2" "Class1" "Class1" "Class1" "Class2" "Class1"
## [473] "Class1" "Class1" "Class2" "Class2" "Class1" "Class2" "Class1" "Class1"
## [481] "Class1" "Class1" "Class1" "Class1" "Class1" "Class1" "Class1" "Class1"
## [489] "Class1" "Class1" "Class1" "Class2" "Class1" "Class1" "Class1" "Class1"
## [497] "Class1" "Class1" "Class1" "Class1"
```

```
table(predict(fit, two_class_data, predict.all = TRUE)$individual[2, ])
```

```
##
## Class1 Class2
##    411     89
```

```
table(predict(fit, two_class_data, predict.all = TRUE)$individual[2, ]) / 500
```

```
##
## Class1 Class2
##  0.822  0.178
```

```
mean(fit$oob.times / 500)
```

```
## [1] 0.368216
```

```
exp(-1)
```

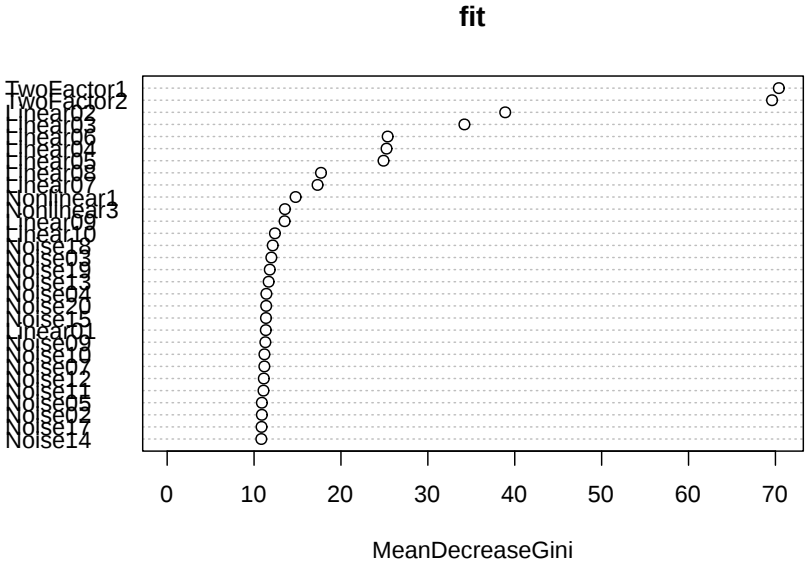
```
## [1] 0.3678794
```

```
fit$importance
```

```
##            MeanDecreaseGini
## TwoFactor1        70.393099
## TwoFactor2        69.608199
## Linear01          11.352368
## Linear02          38.903234
## Linear03          34.207241
## Linear04          25.245839
## Linear05          24.898924
## Linear06          25.374816
## Linear07          17.307651
## Linear08          17.704504
## Linear09          13.516845
## Linear10          12.400794
## Nonlinear1        14.781874
## Nonlinear2         9.795264
## Nonlinear3        13.552729
## Noise01           10.566502
## Noise02           10.898295
## Noise03           11.989430
## Noise04           11.418045
## Noise05           10.899319
## Noise06           10.821215
## Noise07           11.185296
## Noise08           10.604101
## Noise09           11.301260
## Noise10           11.191305
## Noise11           11.084711
## Noise12           11.121672
## Noise13           11.678859
## Noise14           10.835472
## Noise15           11.372321
## Noise16           10.558511
```

```
## Noise17           10.862320
## Noise18           12.148751
## Noise19           11.807009
## Noise20           11.396611
```

```
varImpPlot(fit)
```




```
fit_caret_rf = train(Class ~ ., data = two_class_data,
                     trControl = trainControl(method = "oob"))
```

```
fit_caret_rf
```

```
## Random Forest
##
## 1250 samples
##   35 predictor
##    2 classes: 'Class1', 'Class2'
##
## No pre-processing
## Resampling results across tuning parameters:
##
##   mtry  Accuracy  Kappa
##    2    0.8168    0.6309813
##   18    0.8304    0.6592744
##   35    0.8288    0.6564099
##
```

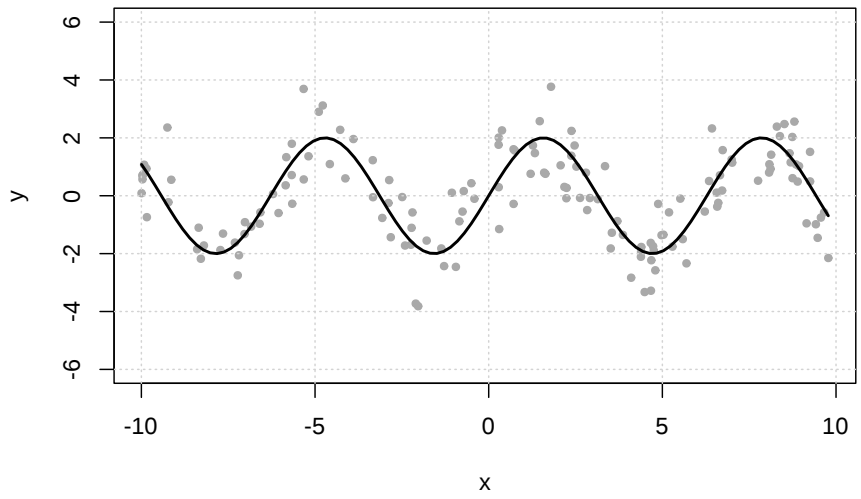
```
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was mtry = 18.
```

15.4 Boosting

```
sin_dgp = function(sample_size = 150) {
  x = runif(n = sample_size, min = -10, max = 10)
  y = 2 * sin(x) + rnorm(n = sample_size)
  tibble(x = x, y = y)
}
```

```
set.seed(42)
sim_data = sin_dgp()
```

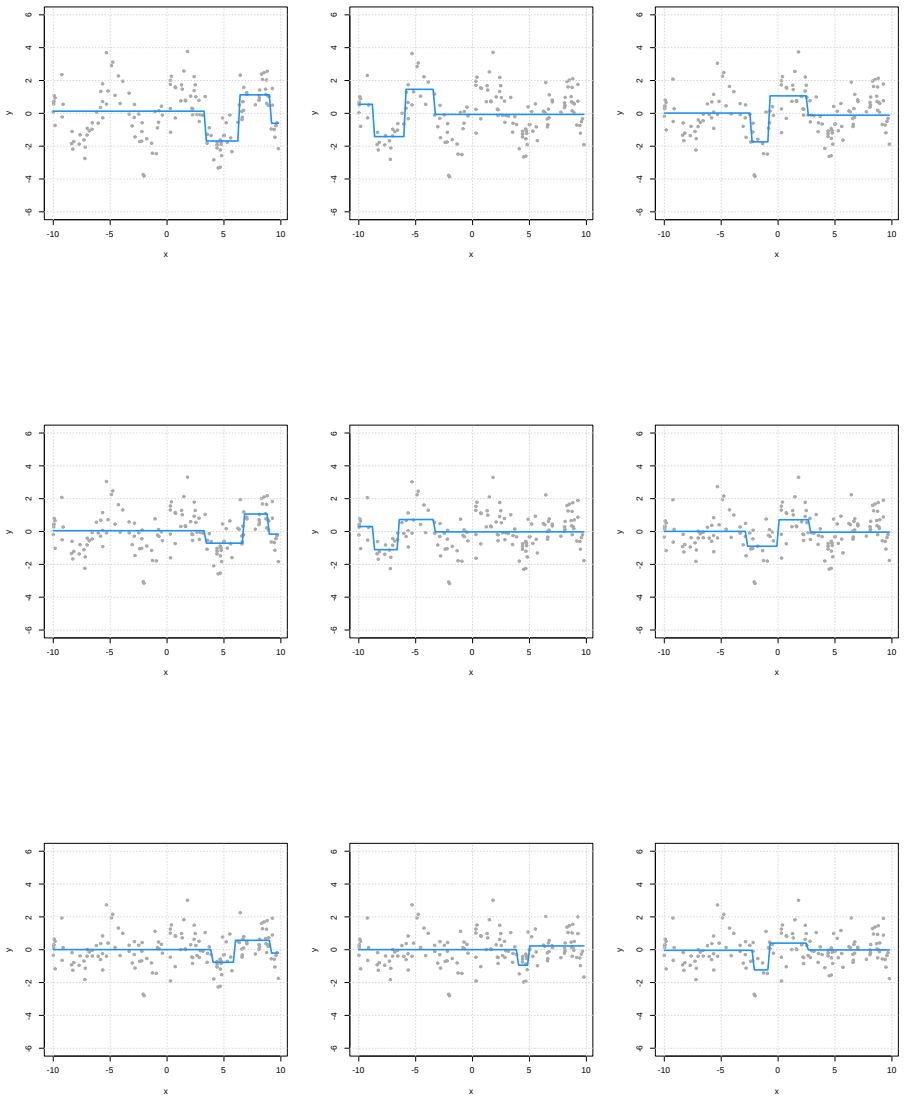
```
plot(sim_data, ylim = c(-6, 6), pch = 20, col = "darkgrey")
grid()
curve(2 * sin(x), col = "black", add = TRUE, lwd = 2)
```

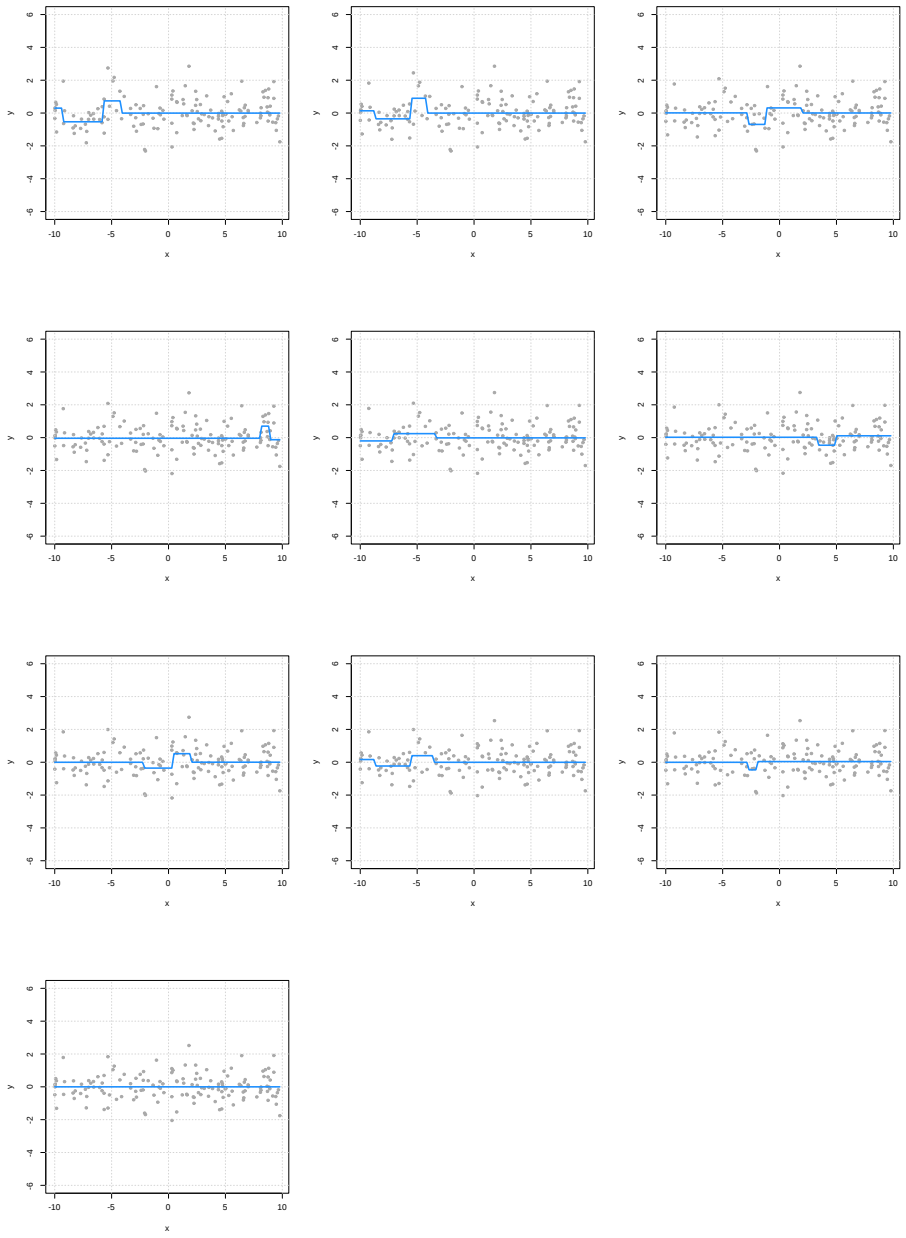



```
sim_data_for_boosting = sim_data
```

```
par(mfrow = c(1, 3))
splits = 99
while(splits > 0) {
  plot(sim_data_for_boosting, ylim = c(-6, 6), pch = 20, col = "darkgrey")
```

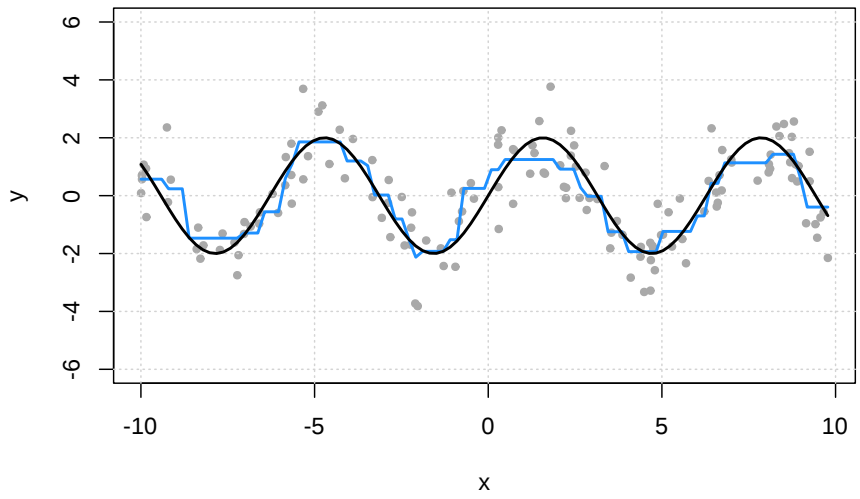
```
  grid()
  fit = rpart(y ~ x, data = sim_data_for_boosting, maxdepth = 2)
  curve(predict(fit, data.frame(x = x)), add = TRUE, lwd = 2, col = "dodgerblue")
  sim_data_for_boosting$y = sim_data_for_boosting$y - 0.4 * predict(fit)
  splits = nrow(fit$frame) - 1
}
```





```
sim_data_for_boosting = sim_data
tree_list = list()

for (i in 1:100) {
  fit = rpart(y ~ x, data = sim_data_for_boosting, maxdepth = 2)
```

```
  tree_list[[i]] = fit
  sim_data_for_boosting$y = sim_data_for_boosting$y - 0.4 * predict(fit)
}

names(tree_list) = 1:100

boost_pred = function(x) {
  apply(t(map_df(tree_list, predict, data.frame(x = x))), 2, function(x) {0.4 * sum(x)})
}

plot(sim_data, ylim = c(-6, 6), pch = 20, col = "darkgrey")
grid()
curve(boost_pred(x), add = TRUE, lwd = 2, col = "dodgerblue")
curve(2 * sin(x), add = TRUE, col = "black", lwd = 2)
```




```
fit_caret_gbm = train(Class ~ ., data = two_class_data,
                      method = "gbm",
                      trControl = trainControl(method = "cv", number = 5),
                      verbose = FALSE)
```

```
fit_caret_gbm
```

```
## Stochastic Gradient Boosting
##
## 1250 samples
```

```
##   35 predictor
##    2 classes: 'Class1', 'Class2'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 1000, 1001, 999, 1000, 1000
## Resampling results across tuning parameters:
##
##   interaction.depth  n.trees  Accuracy   Kappa
##   1                   50      0.7791409  0.5535950
##   1                  100      0.8223827  0.6422082
##   1                  150      0.8384244  0.6751165
##   2                   50      0.8263923  0.6510654
##   2                  100      0.8504245  0.6995372
##   2                  150      0.8496213  0.6980252
##   3                   50      0.8408117  0.6802285
##   3                  100      0.8576086  0.7140311
##   3                  150      0.8640214  0.7270698
##
## Tuning parameter 'shrinkage' was held constant at a value of 0.1
##
## Tuning parameter 'n.minobsinnode' was held constant at a value of 10
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were n.trees = 150, interaction.depth =
##  3, shrinkage = 0.1 and n.minobsinnode = 10.
```

```
fit_caret_xgb = train(Class ~ ., data = two_class_data,
                      method = "xgbTree",
                      trControl = trainControl(method = "cv", number = 5),
                      tuneLength = 2)
```

```
fit_caret_xgb
```

```
## eXtreme Gradient Boosting
##
## 1250 samples
##   35 predictor
##    2 classes: 'Class1', 'Class2'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 1000, 1001, 999, 1000, 1000
## Resampling results across tuning parameters:
##
##   eta  max_depth  colsample_bytree  subsample  nrounds  Accuracy   Kappa
##   0.3  1          0.6               0.5         50      0.8392214  0.6769540
##   0.3  1          0.6               0.5        100      0.8544055  0.7078959
```

```
##   0.3  1          0.6              1.0         50      0.8296052  0.6565702
##   0.3  1          0.6              1.0        100      0.8464150  0.6910691
##   0.3  1          0.8              0.5         50      0.8360308  0.6705396
##   0.3  1          0.8              0.5        100      0.8512021  0.7012276
##   0.3  1          0.8              1.0         50      0.8288276  0.6551720
##   0.3  1          0.8              1.0        100      0.8512214  0.7010765
##   0.3  2          0.6              0.5         50      0.8488278  0.6965017
##   0.3  2          0.6              0.5        100      0.8536214  0.7061904
##   0.3  2          0.6              1.0         50      0.8464213  0.6916100
##   0.3  2          0.6              1.0        100      0.8576247  0.7142662
##   0.3  2          0.8              0.5         50      0.8456597  0.6901581
##   0.3  2          0.8              0.5        100      0.8592150  0.7177094
##   0.3  2          0.8              1.0         50      0.8600023  0.7189731
##   0.3  2          0.8              1.0        100      0.8631991  0.7256871
##   0.4  1          0.6              0.5         50      0.8384118  0.6750451
##   0.4  1          0.6              0.5        100      0.8447894  0.6882491
##   0.4  1          0.6              1.0         50      0.8384277  0.6748621
##   0.4  1          0.6              1.0        100      0.8584119  0.7156206
##   0.4  1          0.8              0.5         50      0.8472086  0.6931865
##   0.4  1          0.8              0.5        100      0.8599926  0.7188617
##   0.4  1          0.8              1.0         50      0.8432149  0.6845350
##   0.4  1          0.8              1.0        100      0.8656152  0.7300746
##   0.4  2          0.6              0.5         50      0.8400150  0.6789671
##   0.4  2          0.6              0.5        100      0.8400117  0.6795340
##   0.4  2          0.6              1.0         50      0.8440118  0.6868629
##   0.4  2          0.6              1.0        100      0.8472470  0.6933817
##   0.4  2          0.8              0.5         50      0.8400278  0.6791399
##   0.4  2          0.8              0.5        100      0.8448117  0.6885384
##   0.4  2          0.8              1.0         50      0.8520055  0.7030056
##   0.4  2          0.8              1.0        100      0.8520119  0.7029730
##
## Tuning parameter 'gamma' was held constant at a value of 0
## Tuning
##  parameter 'min_child_weight' was held constant at a value of 1
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were nrounds = 100, max_depth = 1, eta
##  = 0.4, gamma = 0, colsample_bytree = 0.8, min_child_weight = 1 and subsample
##  = 1.
```

Chapter 16

Practical Issues

16.1 STAT 432 Materials

- Suggested Reading: The `caret` Package: Model Training and Tuning

---

```
library("tidyverse")
library("caret")
library("rpart")
library("rpart.plot")
```

16.2 Feature Scaling

```
# generate date with x1 and x2 on very different scales
gen_reg_data = function(sample_size = 250, beta_1 = 1, beta_2 = 1) {
  x_1 = runif(n = sample_size, min = 0, max = 1)
  x_2 = runif(n = sample_size, min = 0, max = 1000)
  y = beta_1 * x_1 + beta_2 * x_2 + rnorm(n = sample_size)
  tibble(x_1 = x_1, x_2 = x_2, y = y)
}
```

```
# generate train and test sets
set.seed(42)
trn = gen_reg_data(beta_1 = 10, beta_2 = 0.001)
tst = gen_reg_data(beta_1 = 10, beta_2 = 0.001)
```

```
trn %>%
  mutate(x1_scaled = scale(x_1),
         x2_scaled = scale(x_2))
```

```
## # A tibble: 250 x 5
##      x_1   x_2     y x1_scaled[,1] x2_scaled[,1]
##    <dbl> <dbl> <dbl>         <dbl>         <dbl>
##  1 0.915 334.   8.39        1.39          -0.462
##  2 0.937 188.   9.61        1.47          -0.958
##  3 0.286 270.   1.93       -0.765         -0.681
##  4 0.830 531.   9.03        1.10           0.207
##  5 0.642  21.5  7.74        0.455         -1.53
##  6 0.519 799.   4.96        0.0340         1.12
##  7 0.737 110.   6.74        0.780         -1.22
##  8 0.135 540.   1.93       -1.29           0.238
##  9 0.657 571.   6.12        0.507          0.345
## 10 0.705 619.   7.29        0.672          0.507
## # ... with 240 more rows
```

```
# create scaled datasets
scale_trn = preProcess(trn[, 1:2])
trn_scaled = predict(scale_trn, trn)
tst_scaled = predict(scale_trn, tst)
```

```
trn_scaled
```

```
## # A tibble: 250 x 3
##        x_1    x_2     y
##      <dbl>  <dbl> <dbl>
##  1  1.39   -0.462  8.39
##  2  1.47   -0.958  9.61
##  3 -0.765  -0.681  1.93
##  4  1.10    0.207  9.03
##  5  0.455  -1.53   7.74
##  6  0.0340  1.12   4.96
##  7  0.780  -1.22   6.74
##  8 -1.29    0.238  1.93
##  9  0.507   0.345  6.12
## 10  0.672   0.507  7.29
## # ... with 240 more rows
```

```
# linear models -> scaling doesn't matter
# knn -> scaling does matter!
# tress -> scaling doesn't matter
```

```
lm_u = lm(y ~ ., data = trn)
lm_s = lm(y ~ ., data = trn_scaled)
```

```
coef(lm_u)
```

```
##   (Intercept)           x_1           x_2
## -0.0816891578 10.0859803604  0.0009962049
```

```
coef(lm_s)
```

```
## (Intercept)         x_1         x_2
##   5.5221016   2.9395824   0.2928511
```

```
head(cbind(predict(lm_u, tst),
           predict(lm_s, tst_scaled)))
```

```
##        [,1]      [,2]
## 1 9.2073071 9.2073071
## 2 1.3001017 1.3001017
## 3 8.5186045 8.5186045
## 4 5.4156184 5.4156184
## 5 5.6232737 5.6232737
## 6 0.1839076 0.1839076
```

```
all(predict(lm_u, tst) == predict(lm_s, tst_scaled))
```

```
## [1] FALSE
```

```
identical(predict(lm_u, tst), predict(lm_s, tst_scaled))
```

```
## [1] FALSE
```

```
all.equal(predict(lm_u, tst), predict(lm_s, tst_scaled))
```

```
## [1] TRUE
```

```
knn_u = knnreg(y ~ ., data = trn)
knn_s = knnreg(y ~ ., data = trn_scaled)
```

```
all.equal(predict(knn_u, tst), predict(knn_s, tst_scaled))
```

```
## [1] "Mean relative difference: 0.4687973"
```

```
tree_u = rpart(y ~ ., data = trn)
tree_s = rpart(y ~ ., data = trn_scaled)
```

```
identical(predict(tree_u, tst), predict(tree_s, tst_scaled)) #!
```

```
## [1] TRUE
```

```
all.equal(predict(tree_u, tst), predict(tree_s, tst_scaled))
```

```
## [1] TRUE
```

```
set.seed(42)
fit_caret_unscaled = train(
  y ~ .,
  data = trn,
  method = "knn",
  trControl = trainControl(method = "cv", number = 5)
)
fit_caret_unscaled
```

```
## k-Nearest Neighbors
##
## 250 samples
##   2 predictor
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 200, 200, 200, 200, 200
## Resampling results across tuning parameters:
##
##   k  RMSE      Rsquared     MAE
##   5  3.470440  0.018004804  2.851823
##   7  3.261563  0.007334744  2.694560
##   9  3.254494  0.016335313  2.685868
##
## RMSE was used to select the optimal model using the smallest value.
## The final value used for the model was k = 9.
```

```
set.seed(42)
fit_caret_scaled = train(
  y ~ ., data = trn,
  method = "knn",
  preProcess = c("center", "scale"),
  trControl = trainControl(method = "cv", number = 5))
fit_caret_scaled
```

```
## k-Nearest Neighbors
##
## 250 samples
##   2 predictor
##
## Pre-processing: centered (2), scaled (2)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 200, 200, 200, 200, 200
## Resampling results across tuning parameters:
##
##   k  RMSE      Rsquared   MAE
```

```
##   5  1.169505  0.8571820  0.9371406
##   7  1.119903  0.8693966  0.9064097
##   9  1.088236  0.8775694  0.8802256
##
## RMSE was used to select the optimal model using the smallest value.
## The final value used for the model was k = 9.
```

```
predict(fit_caret_scaled, tst[1:10, ])
```

```
##  [1] 8.9527676 1.5439899 8.1698569 5.1921137 5.6422941 0.9428402 5.8606918
##  [8] 7.5202928 3.0247919 8.7029136
```

```
# re-generate train and test sets
set.seed(42)
trn = gen_reg_data(beta_1 = 1, beta_2 = 1)
tst = gen_reg_data(beta_1 = 1, beta_2 = 1)
```

```
set.seed(42)
fit_caret_unscaled = train(
  y ~ .,
  data = trn,
  method = "knn",
  trControl = trainControl(method = "cv", number = 5)
)
fit_caret_unscaled
```

```
## k-Nearest Neighbors
##
## 250 samples
##   2 predictor
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 200, 200, 200, 200, 200
## Resampling results across tuning parameters:
##
##   k  RMSE      Rsquared   MAE
##   5  3.938920  0.9998145  2.835701
##   7  4.664098  0.9997504  3.370510
##   9  5.315904  0.9996846  3.924115
##
## RMSE was used to select the optimal model using the smallest value.
## The final value used for the model was k = 5.
```

```
set.seed(42)
fit_caret_scaled = train(
  y ~ ., data = trn,
  method = "knn",
```

```
  preProcess = c("center", "scale"),
  trControl = trainControl(method = "cv", number = 5))
fit_caret_scaled
```

```
## k-Nearest Neighbors
##
## 250 samples
##   2 predictor
##
## Pre-processing: centered (2), scaled (2)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 200, 200, 200, 200, 200
## Resampling results across tuning parameters:
##
##   k  RMSE      Rsquared   MAE
##   5  27.58391  0.9914624  21.49055
##   7  28.15013  0.9915600  21.19110
##   9  28.90703  0.9914946  22.63816
##
## RMSE was used to select the optimal model using the smallest value.
## The final value used for the model was k = 5.
```

16.3 Categorical Features

```
gen_cat_data = function(sample_size = 250) {

  # generate categorical x data
  x = sample(LETTERS[1:10], size = sample_size, replace = TRUE)

  # generate y data, different means for different categories
  y = case_when(
    x == "A" ~ rnorm(n = sample_size, mean = 1),
    x == "B" ~ rnorm(n = sample_size, mean = 1),
    x == "C" ~ rnorm(n = sample_size, mean = 1),
    x == "D" ~ rnorm(n = sample_size, mean = 5),
    x == "E" ~ rnorm(n = sample_size, mean = 5),
    x == "F" ~ rnorm(n = sample_size, mean = 5),
    x == "G" ~ rnorm(n = sample_size, mean = 11),
    x == "H" ~ rnorm(n = sample_size, mean = 11),
    x == "I" ~ rnorm(n = sample_size, mean = 13),
    x == "J" ~ rnorm(n = sample_size, mean = 13),
  )
```

```
  # return tibble
  tibble(x = x, y = y)
}
```

```
x_levels = data.frame(
  x = LETTERS[1:10]
)
```

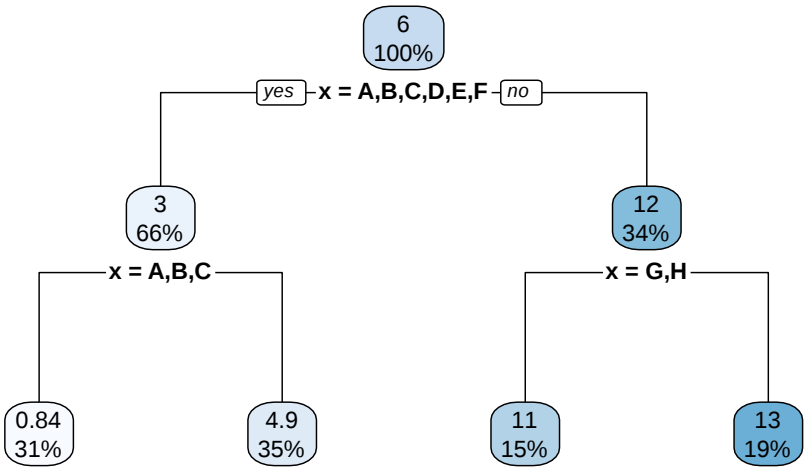
```
cat_trn = gen_cat_data()
head(cat_trn, n = 10)
```

```
## # A tibble: 10 x 2
##    x          y
##    <chr>  <dbl>
##  1 J     12.0
##  2 F      3.49
##  3 G     11.0
##  4 E      2.15
##  5 E      3.36
##  6 C      1.43
##  7 F      4.67
##  8 E      6.18
##  9 I     13.4
## 10 B      0.806
```

```
lm(y ~ x, data = cat_trn)
```

```
##
## Call:
## lm(formula = y ~ x, data = cat_trn)
##
## Coefficients:
## (Intercept)           xB           xC           xD           xE           xF
##     0.66805      0.04599      0.38439      4.38387      4.12645      4.35950
##          xG           xH           xI           xJ
##    10.15284     10.11491     12.27431     12.45589
```

```
rpart.plot(rpart(y ~ x, data = cat_trn))
```




```
knn_cat_mod = knnreg(y ~ x, data = cat_trn, use.all = TRUE)
predict(knn_cat_mod, x_levels)
```

```
##  [1]  0.6680481  0.7140409  1.0524340  5.0519187  4.7944962  5.0275456
##  [7] 10.8208879 10.7829575 12.9423601 13.1239336
```

```
tree_cat_mod = rpart(y ~ x, data = cat_trn, cp = 0)
predict(tree_cat_mod, x_levels)
```

```
##          1          2          3          4          5          6          7
##  0.6680481  0.7140409  1.0524340  5.0519187  4.7944962  5.0275456 10.8208879
##          8          9         10
## 10.7829575 12.9423601 13.1239336
```

```
all.equal(predict(knn_cat_mod, cat_trn), predict(rpart(y ~ x, data = cat_trn, cp = 0)))
```

```
## [1] TRUE
```

```
lm_cat_mod = lm(y ~ x, data = cat_trn)
predict(lm_cat_mod, x_levels)
```

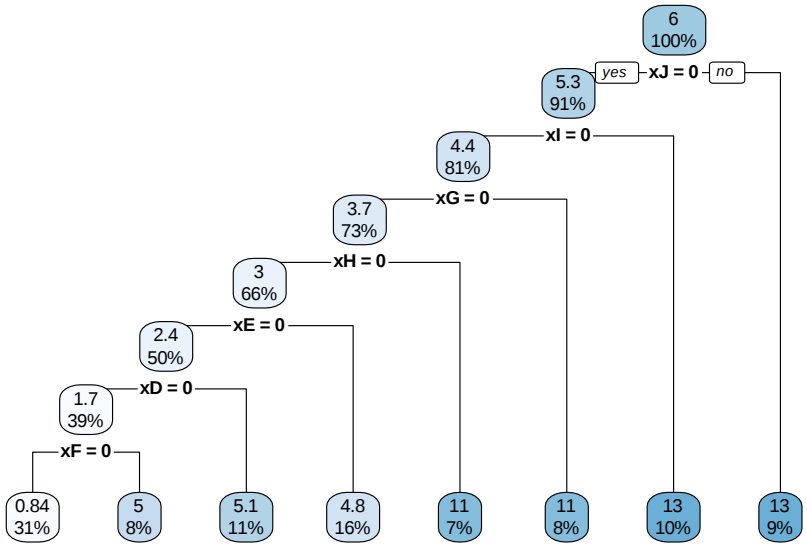
```
##          1          2          3          4          5          6          7
##  0.6680481  0.7140409  1.0524340  5.0519187  4.7944962  5.0275456 10.8208879
##          8          9         10
## 10.7829575 12.9423601 13.1239336
```

```
dum_trn = as_tibble(predict(dummyVars(~., data = cat_trn), cat_trn))
```

```
lm(y ~ . + 0, data = dum_trn)
```

```
##
## Call:
## lm(formula = y ~ . + 0, data = dum_trn)
##
## Coefficients:
##     xA      xB      xC      xD      xE      xF      xG      xH      xI      xJ
##  0.668   0.714   1.052   5.052   4.794   5.028  10.821  10.783  12.942  13.124
```

```
rpart.plot(rpart(y ~ ., data = dum_trn))
```



Part IV

Mathematics

Chapter 17

Introduction

asdf

Part V

Computing

Chapter 18

Introduction

asdf

Chapter 19

Resources

This is not a book about R. It is however, a book that uses R. Because of this, you will need to be familiar with R. The text will point out some thing about R along the way, but some previous study of R is necessary.

The following (freely available) readings are highly recommended:

- Hands-On Programming with R - *Garrett Grolemund*
  - If you have never used R or RStudio before, Part 1, Chapters 1 - 3, will be useful.
- R for Data Science - *Garrett Grolemund, Hadley Wickham*
  - This book helps getting you up to speed working with data in R. While it is a lot of reading, Chapters 1 - 21 are highly recommended.
- Advanced R - *Hadley Wickham*
  - Part I, Chapters 1 - 8, of this book will help create a mental model for working with R. These chapters are not an easy read, so they should be returned to often. (Chapter 2 could be safely skipped for our purposes, but is important if you will use R in the long term.)

If you are a UIUC student who took the course STAT 420, the first six chapters of that book could serve as a nice refresher.

- Applied Statistics with R - *David Dalpiaz*

19.1 Resources

The following resources are more specific or more advanced, but could still prove to be useful.

19.1.1 R

- Efficient R programming
- R Programming for Data Science
- R Graphics Cookbook
- Modern Dive
- The `tidyverse` Website
  - `dplyr` Website
  - `readr` Website
  - `tibble` Website
  - `forcats` Website

19.1.2 RStudio

- RStudio IDE Cheatsheet
- RStudio Resources

19.1.3 R Markdown

- R Markdown Cheatsheet
- R Markdown: The Definitive Guide - *Yihui Xie, J. J. Allaire, Garrett Grolemund*
- R4DS R Markdown Chapter

19.1.3.1 Markdown

- Daring Fireball - Markdown: Basics
- GitHub - Mastering Markdown
- CommonMark

19.2 BSL Idioms

Things here supersede everything above.

19.2.1 Reference Style

- `tidyverse` Style Guide

19.2.2 BSL Style Overrides

- TODO: = instead of <-
  - http://thecoatlessprofessor.com/programming/an-opinionated-tale-of-why-you-should-replace---with-/
- TODO: never use T or F, only TRUE or FALSE

```
FALSE == TRUE
```

```
## [1] FALSE
```

```
F     == TRUE
```

```
## [1] FALSE
```

```
F     =  TRUE
F     == TRUE
```

```
## [1] TRUE
```

- TODO: never ever ever use attach()
- TODO: never ever ever use <<-
- TODO: never ever ever use setwd() or set a working directory some other way
- TODO: a newline before and after any chunk
- TODO: use headers appropriately! (short names, good structure)
- TODO: never ever ever put spaces in filenames. use -. (others will use _.)
- TODO: load all needed packages at the beginning of an analysis in a single chunk (TODO: pros and cons of this approach)
- TODO: one plot per chunk! no other printed output

Be consistent...

- with yourself!
- with your group!
- with your organization!

```
set.seed(1337);mu=10;sample_size=50;samples=100000;
x_bars=rep(0, samples)
for(i in 1:samples)
{
x_bars[i]=mean(rpois(sample_size,lambda = mu))}
x_bar_hist=hist(x_bars,breaks=50,main="Histogram of Sample Means",xlab="Sample Means",col="darkor
mean(x_bars>mu-2*sqrt(mu)/sqrt(sample_size)&x_bars<mu+2*sqrt(mu)/sqrt(sample_size))
```

19.2.3 Objects and Functions

To understand computations in R, two slogans are helpful:

- Everything that exists is an object.
- Everything that happens is a function call.

— John Chambers

- TODO: Functions + Objects
  - these are the inputs and outputs of functions:
    - functions
    - vectors
    - lists
    - tibbles (dfs)

19.2.4 Print versus Return

```
cars_mod = lm(dist ~ speed, data = cars)
```

```
summary(cars_mod)
```

```
##
## Call:
## lm(formula = dist ~ speed, data = cars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -29.069  -9.525  -2.272   9.215  43.201
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -17.5791     6.7584  -2.601   0.0123 *
## speed         3.9324     0.4155   9.464 1.49e-12 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 15.38 on 48 degrees of freedom
## Multiple R-squared:  0.6511, Adjusted R-squared:  0.6438
## F-statistic: 89.57 on 1 and 48 DF,  p-value: 1.49e-12
```

```
is.list(summary(cars_mod))
```

```
## [1] TRUE
```

```
names(summary(cars_mod))
```

```
##  [1] "call"          "terms"         "residuals"     "coefficients"
##  [5] "aliased"       "sigma"         "df"            "r.squared"
##  [9] "adj.r.squared" "fstatistic"    "cov.unscaled"
```

```
str(summary(cars_mod))
```

```
## List of 11
```

```
##  $ call         : language lm(formula = dist ~ speed, data = cars)
##  $ terms        :Classes 'terms', 'formula'  language dist ~ speed
##   .. ..- attr(*, "variables")= language list(dist, speed)
##   .. ..- attr(*, "factors")= int [1:2, 1] 0 1
##   .. .. ..- attr(*, "dimnames")=List of 2
##   .. .. .. ..$ : chr [1:2] "dist" "speed"
##   .. .. .. ..$ : chr "speed"
##   .. ..- attr(*, "term.labels")= chr "speed"
##   .. ..- attr(*, "order")= int 1
##   .. ..- attr(*, "intercept")= int 1
##   .. ..- attr(*, "response")= int 1
##   .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##   .. ..- attr(*, "predvars")= language list(dist, speed)
##   .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
##   .. .. ..- attr(*, "names")= chr [1:2] "dist" "speed"
##  $ residuals    : Named num [1:50] 3.85 11.85 -5.95 12.05 2.12 ...
##   ..- attr(*, "names")= chr [1:50] "1" "2" "3" "4" ...
##  $ coefficients : num [1:2, 1:4] -17.579 3.932 6.758 0.416 -2.601 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:2] "(Intercept)" "speed"
##   .. ..$ : chr [1:4] "Estimate" "Std. Error" "t value" "Pr(>|t|)"
##  $ aliased      : Named logi [1:2] FALSE FALSE
##   ..- attr(*, "names")= chr [1:2] "(Intercept)" "speed"
##  $ sigma        : num 15.4
##  $ df           : int [1:3] 2 48 2
##  $ r.squared    : num 0.651
##  $ adj.r.squared: num 0.644
##  $ fstatistic   : Named num [1:3] 89.6 1 48
##   ..- attr(*, "names")= chr [1:3] "value" "numdf" "dendf"
##  $ cov.unscaled : num [1:2, 1:2] 0.19311 -0.01124 -0.01124 0.00073
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:2] "(Intercept)" "speed"
##   .. ..$ : chr [1:2] "(Intercept)" "speed"
##  - attr(*, "class")= chr "summary.lm"
```

```
# RStudio only
View(summary(cars_mod))
```

19.2.5 Help

- TODO: ?, google, stack overflow, (office hours, course forums)

19.2.6 Keyboard Shortcuts

- TODO: copy-paste, switch program, switch tab, etc…
- TODO: TAB!!!
- TODO: new chunk!
- TODO: style!
- TODO: keyboard shortcut for keyboard shortcut

19.3 Common Issues

- TODO: cannot find function called ””

- TODO: https://stat545.com/
- TODO: https://atrebas.github.io/post/2019-01-15-2018-learning/
- TODO: https://www.burns-stat.com/pages/Tutor/R_inferno.pdf

Chapter 20

Simulation and Bootstrap

20.1 STAT 432 Materials

- **Slides** | Bootstrap

Note: In the future this should become two chapters, and be greatly expanded. Perhaps introduce simulation, ECDF, and plug-in-principle very early.

```
library(microbenchmark)
```

```
# create some data
set.seed(1)
data = rexp(n = 100, rate = 0.5)
```

```
# quantities of interest
1 / 0.5 # mean
```

```
## [1] 2
```

```
qexp(0.5, rate = 0.5)
```

```
## [1] 1.386294
```

```
pexp(8, rate = 0.5, lower.tail = FALSE) # probability P[X > 8]
```

```
## [1] 0.01831564
```

```
# estimates using sampled data
mean(data) # boring
```

```
## [1] 2.061353
```

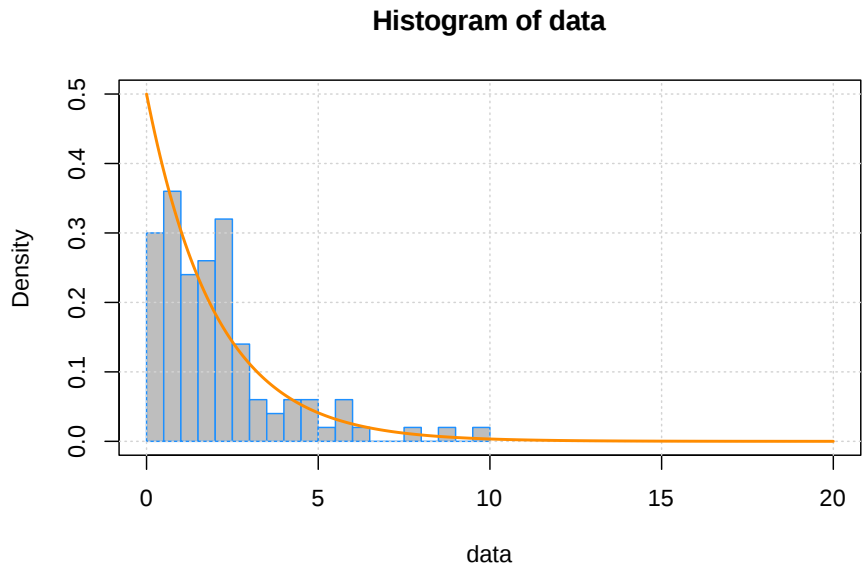
```
median(data)
```

```
## [1] 1.617539
```

```
mean(data > 8)
```

```
## [1] 0.02
```

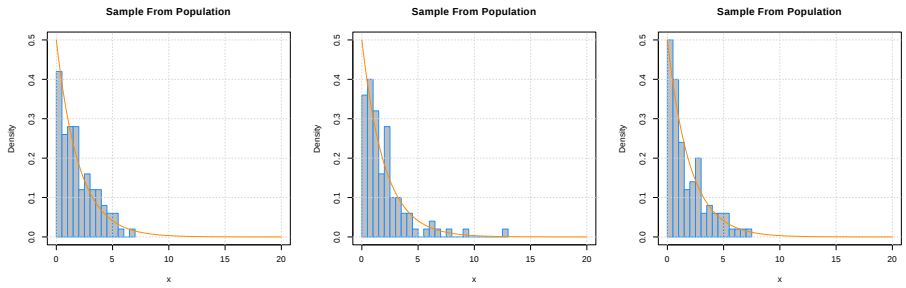
```
# histogram of data with population distribution
hist(data,
     border = "dodgerblue", col = "grey", breaks = 20,
     probability = TRUE, ylim = c(0, 0.5), xlim = c(0, 20))
box()
grid()
curve(dexp(x, rate = 0.5), add = TRUE, col = "darkorange", n = 250, lwd = 2)
```




```
# define graphical parameters
ylim = c(0, 0.5)
xlim = c(0, 20)
```

```
# resampling from the true distribution
par(mfrow = c(1, 3))

hist(rexp(n = 100, rate = 0.5),
     border = "dodgerblue", col = "grey", breaks = 20, probability = TRUE,
     ylim = ylim, xlim = xlim,
```

```
     main = "Sample From Population", xlab = "x")
box()
grid()
curve(dexp(x, rate = 0.5), add = TRUE, col = "darkorange")

hist(rexp(n = 100, rate = 0.5),
     border = "dodgerblue", col = "grey", breaks = 20, probability = TRUE,
     ylim = ylim, xlim = xlim,
     main = "Sample From Population", xlab = "x")
box()
grid()
curve(dexp(x, rate = 0.5), add = TRUE, col = "darkorange")

hist(rexp(n = 100, rate = 0.5),
     border = "dodgerblue", col = "grey", breaks = 20, probability = TRUE,
     ylim = ylim, xlim = xlim,
     main = "Sample From Population", xlab = "x")
box()
grid()
curve(dexp(x, rate = 0.5), add = TRUE, col = "darkorange")
```




```
qexp(0.5, rate = 0.5) # true populaiton median
```

```
## [1] 1.386294
```

```
median(data) # sample median
```

```
## [1] 1.617539
```

```
# simulating medians
# - simulate from known distribution
# - calculate median on simualted data
# - store result
simulated_medians = replicate(n = 10000, median(rexp(n = 100, rate = 0.5)))
```

```
# use EDCD and plug-in principle to learn about distribution and make estimates
mean(simulated_medians)
```

```
## [1] 1.397316
```

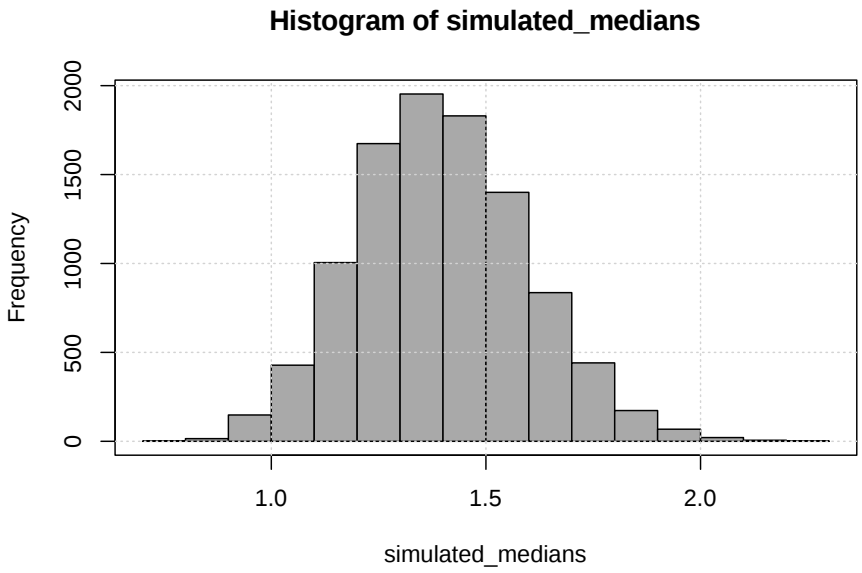
```
sd(simulated_medians)
```

```
## [1] 0.1993581
```

```
quantile(simulated_medians, probs = c(0.025, 0.975))
```

```
##     2.5%    97.5%
## 1.029307 1.807971
```

```
# plot of ECDF of sample median
hist(simulated_medians, col = "darkgrey")
box()
grid()
```




```
pexp(8, rate = 0.5, lower.tail = FALSE) # true probability P[X > 8]
```

```
## [1] 0.01831564
```

```
mean(data > 8) # estimate using ECDF of sample data, mean(I(x > 8))
```

```
## [1] 0.02
```

```
# simulating estimates of P[X > 8]
# - simulate from known distribution
# - calculate proportion of obs greater than 8 in simualted data, mean(I(x > 8))
# - store result
simulated_probs = replicate(n = 10000, mean(rexp(n = 100, rate = 0.5) > 8))
```

```r
# use EDCD and plug-in principle to learn about distribution and make estimates
mean(simulated_probs)
```

```
## [1] 0.018517
```

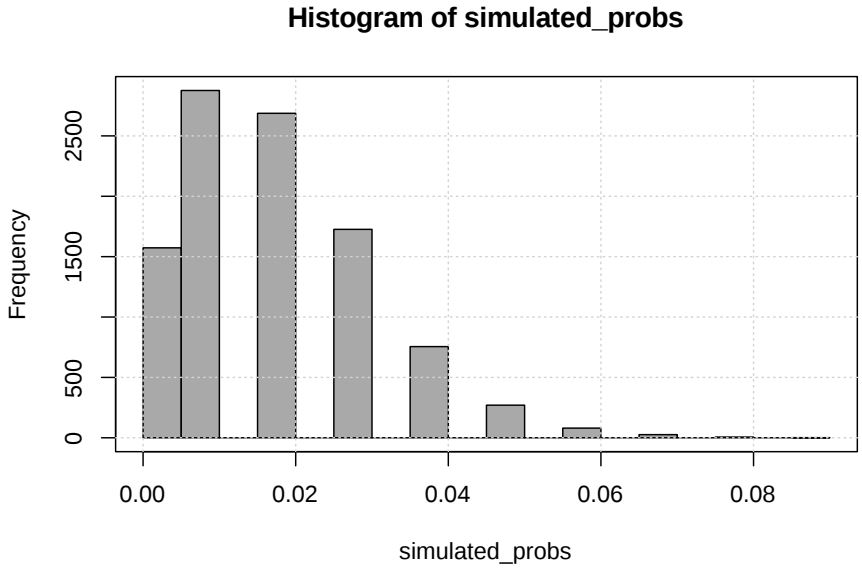
```r
sd(simulated_probs)
```

```
## [1] 0.01353547
```

```r
quantile(simulated_probs, probs = c(0.025, 0.975))
```

```
##  2.5% 97.5%
##  0.00  0.05
```

```r
# plot of ECDF of sample median
hist(simulated_probs, col = "darkgrey")
box()
grid()
```

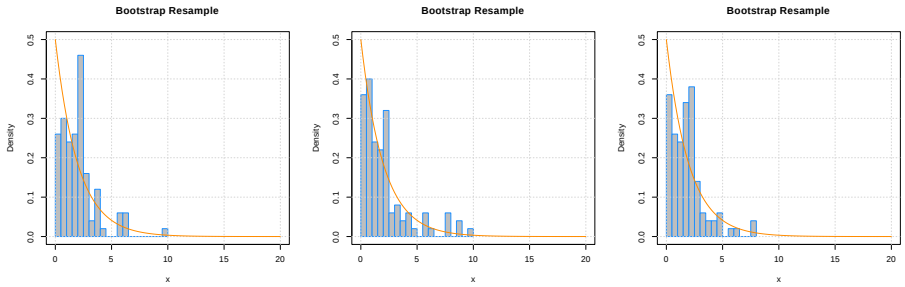



```r
# bootstrap resampling
sample(x = data, replace = TRUE)
```

```
##   [1] 0.29409198 4.35754514 5.51848758 2.47520710 0.89490379 2.21187254
##   [7] 2.64093586 3.14397369 5.78993707 0.27959052 2.36328556 1.18923530
##  [13] 2.78147026 0.41973316 1.52405971 4.56770695 3.56953081 1.62907161
##  [19] 0.29141345 1.54837553 0.21214525 4.72903051 1.13173105 4.35754514
##  [25] 1.95479161 0.26514281 2.15976227 2.36328556 0.70374100 0.60348187
```

```
## [31] 0.17934816 2.46758302 1.60834182 5.51848758 1.45042861 1.50308538
## [37] 0.52165649 6.43557804 1.54837553 1.18923530 1.15742493 9.66562549
## [43] 2.46758302 0.60256599 0.89490379 0.89490379 0.11852241 0.07453705
## [49] 4.20075446 1.07936568 2.04545175 3.56953081 0.11852241 0.77757354
## [55] 4.35754514 0.57118197 3.14397369 0.61889571 2.10908633 0.58824078
## [61] 1.45042861 1.28378518 1.54837553 7.91786570 0.47005490 1.17695944
## [67] 5.78993707 1.62907161 0.40702070 0.47005490 9.66562549 0.77757354
## [73] 1.02834859 1.17695944 2.15976227 1.67401298 0.11887832 0.70374100
## [79] 2.27366283 1.51036367 1.98911158 0.52165649 0.57118197 1.30949327
## [85] 2.50621071 1.54837553 2.04545175 2.47520710 2.78147026 0.21214525
## [91] 0.26514281 1.98911158 1.10928280 0.57118197 0.57118197 7.91786570
## [97] 0.57118197 7.91786570 1.62907161 3.14397369
```

```r
# bootstrap resample from original data
par(mfrow = c(1, 3))

hist(sample(x = data, replace = TRUE),
     border = "dodgerblue", col = "grey", breaks = 20, probability = TRUE,
     ylim = ylim, xlim = xlim,
     main = "Bootstrap Resample", xlab = "x")
box()
grid()
curve(dexp(x, rate = 0.5), add = TRUE, col = "darkorange")

hist(sample(x = data, replace = TRUE),
     border = "dodgerblue", col = "grey", breaks = 20, probability = TRUE,
     ylim = ylim, xlim = xlim,
     main = "Bootstrap Resample", xlab = "x")
box()
grid()
curve(dexp(x, rate = 0.5), add = TRUE, col = "darkorange")

hist(sample(x = data, replace = TRUE),
     border = "dodgerblue", col = "grey", breaks = 20, probability = TRUE,
     ylim = ylim, xlim = xlim,
     main = "Bootstrap Resample", xlab = "x")
box()
grid()
curve(dexp(x, rate = 0.5), add = TRUE, col = "darkorange")
```




```
qexp(0.5, rate = 0.5) # true populaiton median
```

```
## [1] 1.386294
```

```
median(data) # sample median
```

```
## [1] 1.617539
```

```
# boostrapping medians
# - sample with replacement from sample data
# - calculate median on resampled data
# - store result
replicated_medians = replicate(n = 10000, median(sample(data, replace = TRUE)))
```

```
# use EDCD and plug-in principle to learn about distribution and make estimates
mean(replicated_medians)
```

```
## [1] 1.626611
```

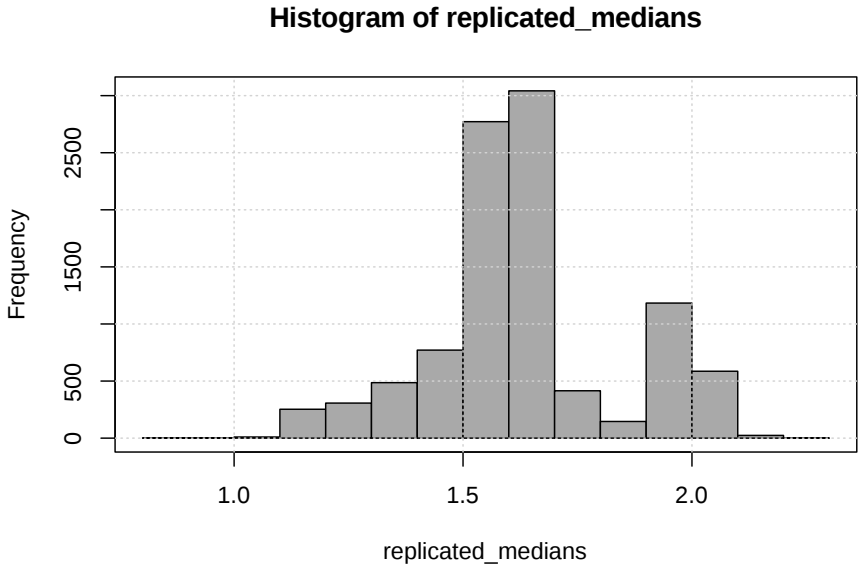
```
sd(replicated_medians)
```

```
## [1] 0.2100116
```

```
# a 95% CI for the true median
quantile(replicated_medians, probs = c(0.025, 0.975))
```

```
##     2.5%    97.5%
## 1.189235 2.045452
```

```
# plot of ECDF of sample median
hist(replicated_medians, col = "darkgrey")
box()
grid()
```




```r
pexp(8, rate = 0.5, lower.tail = FALSE) # true probability P[X > 8]
```

```
## [1] 0.01831564
```

```r
mean(data > 8) # estimate using ECDF of sample data, mean(I(x > 8))
```

```
## [1] 0.02
```

```r
# boostrapping mean(I(x > 8))
# - sample with replacement from sample data
# - calculate proportion of obs greater than 8 in simualted data, mean(I(x > 8))
# - store result
replicated_probs = replicate(n = 10000, mean(sample(data, replace = TRUE) > 8))
```

```r
# use EDCD and plug-in principle to learn about distribution and make estimates
mean(replicated_probs)
```

```
## [1] 0.019998
```

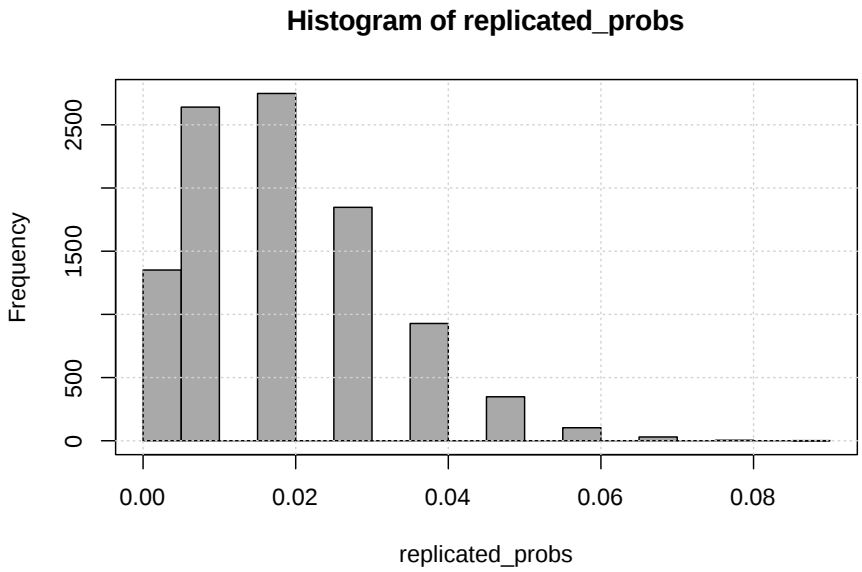
```r
sd(replicated_probs)
```

```
## [1] 0.01390249
```

```r
# a 95% CI for the true P[X > 8]
quantile(replicated_probs, probs = c(0.025, 0.975))
```

```
##  2.5% 97.5%
##  0.00  0.05
```

```
# plot of ECDF of sample median
hist(replicated_probs, col = "darkgrey")
box()
grid()
```




```
# sample from exponential with mean 2
# generate bootstrap replicates of the median
# calculate 95% confidence interval use percentile (quantile) method
# check if true median is in confidence interval each time (hope for ~95%!)
true_median = qexp(0.5, rate = 0.5) # median
```

```
check_if_in_interval = function() {
  data = rexp(n = 100, rate = 0.5)
  replicated_medians = replicate(n = 2000, median(sample(data, replace = TRUE)))
  interval = quantile(replicated_medians, probs = c(0.025, 0.975))
  interval[1] < true_median & true_median < interval[2]
}
```

```
median_in_out = replicate(n = 200, check_if_in_interval())
mean(median_in_out)
```

```
## [1] 0.925
```

```
microbenchmark(check_if_in_interval())
```

```
## Unit: milliseconds
##                    expr      min       lq     mean   median       uq      max
```

```
##  check_if_in_interval() 110.9175 116.2226 125.9969 119.6187 122.2261 298.3749
##  neval
##    100
```

20.2 Misc Notes

- TODO: https://statistics.stanford.edu/sites/g/files/sbiybj6031/f/BIO%2083.pdf

Chapter 21

Data Manipulation

```
library("tidyverse")
```

21.1 dplyr

- The tidyverse Website
- dplyr Website
- dplyr Cheat Sheet
- R4DS: Data Transformation
- R4DS: Pipes

21.2 data.table

- data.table Wiki
- data.table Website
- data.table Cheat Sheet

21.3 Data Splitting with dplyr::anti_join

```
set.seed(42)
# simulate an estimation and validation dataset
```

```
sim_est = as_tibble(caret::twoClassSim(n = 100))
sim_val = as_tibble(caret::twoClassSim(n = 100))
```

```
# merge the rows of the estimation and validation datasets
# to create a training dataset
sim_trn = sim_est %>% bind_rows(sim_val)
```

```
# re-split the data to get better proportions, 80-20 split
sim_est = sim_trn %>% sample_frac(0.8)
sim_val = sim_trn %>% anti_join(sim_est)
```

```
# split datasets together are not the training dataset (check for order)
identical(bind_rows(sim_est, sim_val), sim_trn)
```

```
## [1] FALSE
```

```
# bind_rows(sim_est, sim_val)[1, ] == sim_trn[1, ]
```

```
# split datasets together contain the same observations as the training dataset
setequal(bind_rows(sim_est, sim_val), sim_trn)
```

```
## [1] TRUE
```

Part VI

Analysis

Chapter 22

Introduction

asdf

Part VII

Appendix

Chapter 23

Introduction

Some of the "chapters" in the Appendix are really just chapters that don't have a home in a section yet.

Chapter 24

Additional Reading

24.1 Books

- An Introduction to Statistical Learning
  - Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani
- The Elements of Statistical Learning
  - Trevor Hastie, Robert Tibshirani, and Jerome Friedman
- Mathematics for Machine Learning
  - Marc Peter Deisenroth, A. Aldo Faisal, and Cheng Soon Ong
- Understanding Machine Learning: From Theory to Algorithms
  - Shai Shalev-Shwartz and Shai Ben-David
- Foundations of Machine Learning
  - Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar
- The `caret` Package
  - Max Kuhn
- Feature Engineering and Selection: A Practical Approach for Predictive Models
  - Max Kuhn and Kjell Johnson
- Applied Predictive Modeling
  - Max Kuhn and Kjell Johnson
- Machine Learning: A Probabilistic Perspective
  - Kevin Murphy
- Probability for Statistics and Machine Learning
  - Anirban DasGupta
- From Linear Models to Machine Learning
  - Norman Matloff

24.2 Papers

- Do we Need Hundreds of Classifiers to Solve Real WorldClassification Problems?
  - Manuel Fernandez-Delgado, Eva Cernadas, Senen Barro, Dinani Amorim
- Statistical Modeling: The Two Cultures
  - Leo Breiman
- 50 Years of Data Science
  - David Donoho

24.3 Blog Posts

- Peekaboo: Don't cite the No Free Lunch Theorem

24.4 Miscellaneous

- Machine Learning verus Statistics, A Glossar
  - Rob Tibshirani

Chapter 25

Class Imbalance

```
library("tidyverse")
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.2.1 --
```

```
## v ggplot2 3.2.1     v purrr   0.3.3
## v tibble  2.1.3     v dplyr   0.8.3
## v tidyr   1.0.0     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.4.0
```

```
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```
library("caret")
```

```
## Loading required package: lattice
```

```
##
## Attaching package: 'caret'
```

```
## The following object is masked from 'package:purrr':
##
##     lift
```

```
library("DMwR")
```

```
## Loading required package: grid
```

```
## Registered S3 method overwritten by 'xts':
##   method     from
##   as.zoo.xts zoo
```

```
## Registered S3 method overwritten by 'quantmod':
```

```
##   method            from
##   as.zoo.data.frame zoo
```

```
library("ROSE")
```

```
## Loaded ROSE 0.0-3
```

```
library("randomForest")
```

```
## randomForest 4.6-14
```

```
## Type rfNews() to see new features/changes/bug fixes.
```

```
##
## Attaching package: 'randomForest'
```

```
## The following object is masked from 'package:dplyr':
##
##     combine
```

```
## The following object is masked from 'package:ggplot2':
##
##     margin
```

```
set.seed(42)
trn = as_tibble(twoClassSim(1000,  intercept = -22))
tst = as_tibble(twoClassSim(10000, intercept = -22))
```

```
head(trn, n = 10)
```

```
## # A tibble: 10 x 16
##    TwoFactor1 TwoFactor2 Linear01 Linear02 Linear03 Linear04 Linear05 Linear06
##         <dbl>      <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>
##  1      0.386     3.14      0.251   -0.686   -0.142   0.0712    0.173    1.42
##  2     -1.04     -0.415    -0.278   -0.793   -0.814   0.970    -1.27     0.557
##  3     -0.108     1.04     -1.72    -0.407   -0.326   0.310    -0.868    0.981
##  4      0.590     1.04     -2.01    -1.15     0.378  -0.140     0.626   -0.586
##  5      1.11     -0.0699   -1.29     1.12    -1.99   -0.326    -0.106    0.939
##  6      0.217    -0.490     0.366   -0.879   -0.999  -0.119    -0.256   -0.0647
##  7      1.84      2.04     -0.152    1.28     0.284   0.894    -0.719   -0.254
##  8      1.61     -1.85     -0.734   -1.45    -1.10    0.211    -2.02    -0.867
##  9      3.09      2.09     -0.782    0.842    0.983  -0.489     0.390   -2.21
## 10     -0.553     0.392     0.552   -0.603   -1.02   -0.220    -2.39    -0.915
## # ... with 8 more variables: Linear07 <dbl>, Linear08 <dbl>, Linear09 <dbl>,
## #   Linear10 <dbl>, Nonlinear1 <dbl>, Nonlinear2 <dbl>, Nonlinear3 <dbl>,
## #   Class <fct>
```

```
table(trn$Class)
```

```
##
## Class1 Class2
```

```
##    973     27
```

```
trn_down = downSample(
  x = trn[, -ncol(trn)],
  y = trn$Class
)

trn_up = upSample(
  x = trn[, -ncol(trn)],
  y = trn$Class
)

trn_rose = ROSE(Class ~ ., data = trn)$data
```

```
table(trn_down$Class)
```

```
##
## Class1 Class2
##     27     27
```

```
table(trn_up$Class)
```

```
##
## Class1 Class2
##    973    973
```

```
table(trn_rose$Class)
```

```
##
## Class1 Class2
##    514    486
```

```
cv = trainControl(method = "cv", number = 5)
cv_down = trainControl(method = "cv", number = 5, sampling = "down")
cv_up = trainControl(method = "cv", number = 5, sampling = "up")
cv_rose = trainControl(method = "cv", number = 5, sampling = "rose")
```

```
mod = train(
  Class ~ .,
  data = trn,
  method = "rf",
  trControl = cv
)

mod_down = train(
  Class ~ .,
  data = trn,
  method = "rf",
  trControl = cv_down
```

```
)

mod_up = train(
  Class ~ .,
  data = trn,
  method = "rf",
  trControl = cv_up
)

mod_rose = train(
  Class ~ .,
  data = trn,
  method = "rf",
  trControl = cv_rose
)
```

```
mod_rose
```

```
## Random Forest
##
## 1000 samples
##   15 predictor
##    2 classes: 'Class1', 'Class2'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 800, 801, 800, 799, 800
## Addtional sampling using ROSE
##
## Resampling results across tuning parameters:
##
##   mtry  Accuracy   Kappa
##    2    0.9560146  0.4410583
##    8    0.9589745  0.5220785
##   15    0.9459944  0.3648353
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was mtry = 8.
```

```
confusionMatrix(mod_rose)
```

```
## Cross-Validated (5 fold) Confusion Matrix
##
## (entries are percentual average cell counts across resamples)
##
##           Reference
```

```
## Prediction Class1 Class2
##     Class1   93.7    0.5
##     Class2    3.6    2.2
##
##  Accuracy (average) : 0.959
```

```
confusionMatrix(
  data = predict(mod_rose, tst),
  reference = tst$Class
)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction Class1 Class2
##     Class1   9093     25
##     Class2    632    250
##
##                Accuracy : 0.9343
##                  95% CI : (0.9293, 0.9391)
##     No Information Rate : 0.9725
##     P-Value [Acc > NIR] : 1
##
##                   Kappa : 0.4073
##
##  Mcnemar's Test P-Value : <2e-16
##
##             Sensitivity : 0.9350
##             Specificity : 0.9091
##          Pos Pred Value : 0.9973
##          Neg Pred Value : 0.2834
##              Prevalence : 0.9725
##          Detection Rate : 0.9093
##    Detection Prevalence : 0.9118
##       Balanced Accuracy : 0.9221
##
##        'Positive' Class : Class1
##
```

```
rf_mod = randomForest(
  Class ~ ., data = trn,
  strata = trn$Class,
  sampsize = c(1, 1),
  mtry = 1,
  ntree = 2500
)
rf_mod
```

```
##
## Call:
##  randomForest(formula = Class ~ ., data = trn, strata = trn$Class,      sampsize = c
##                Type of random forest: classification
##                      Number of trees: 2500
## No. of variables tried at each split: 1
##
##         OOB estimate of  error rate: 3.5%
## Confusion matrix:
##        Class1 Class2 class.error
## Class1    956     17  0.01747174
## Class2     18      9  0.66666667
```

```
predict(rf_mod) == "Class2"
```

```
##    [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##   [13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##   [25] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##   [37] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##   [49] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE
##   [61] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##   [73] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##   [85] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##   [97] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [109] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [121] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [133] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [145] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [157] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [169] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
##  [181] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [193] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
##  [205] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
##  [217] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [229] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [241] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [253] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [265] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [277] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE
##  [289] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [301] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [313] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [325] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [337] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [349] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [361] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
##  [373] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
##  [385] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [397] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [409] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [421] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [433] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [445]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [457] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
##  [469] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
##  [481] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [493] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [505] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [517] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [529] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [541] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [553] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [565]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [577] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [589] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [601] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
##  [613] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [625] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
##  [637] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [649] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [661] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [673] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [685] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [697] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [709] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [721] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
##  [733] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
##  [745] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [757] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [769] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [781] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [793] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [805] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [817] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [829] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [841] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [853] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [865] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
##  [877] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [889] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [901] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [913]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
##  [925] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [937] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
##  [949] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [961] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [973] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [985] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [997] FALSE FALSE FALSE FALSE
```

```
rf_mod
```

```
##
## Call:
##  randomForest(formula = Class ~ ., data = trn, strata = trn$Class,      sampsize = c
##                Type of random forest: classification
##                      Number of trees: 2500
## No. of variables tried at each split: 1
##
##         OOB estimate of  error rate: 3.5%
## Confusion matrix:
##        Class1 Class2 class.error
## Class1    956     17  0.01747174
## Class2     18      9  0.66666667
```

Chapter 26

Missing Data

```
library("tidyverse")
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.2.1 --
```

```
## v ggplot2 3.2.1     v purrr   0.3.3
## v tibble  2.1.3     v dplyr   0.8.3
## v tidyr   1.0.0     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.4.0
```

```
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```
library("caret")
```

```
## Loading required package: lattice
```

```
##
## Attaching package: 'caret'
```

```
## The following object is masked from 'package:purrr':
##
##     lift
```

```
airquality = as_tibble(airquality)
```

```
# airquality %>% View()
```

```
count_na = function(x) {
  sum(is.na(x))
}
```

```
apply(airquality, 2, count_na)
```

```
##   Ozone Solar.R    Wind    Temp   Month     Day
##      37       7       0       0       0       0
```

```
aq = airquality %>%
  filter(!is.na(Ozone))
```

```
aq
```

```
## # A tibble: 116 x 6
##    Ozone Solar.R  Wind  Temp Month   Day
##    <int>   <int> <dbl> <int> <int> <int>
##  1    41     190   7.4    67     5     1
##  2    36     118   8      72     5     2
##  3    12     149  12.6    74     5     3
##  4    18     313  11.5    62     5     4
##  5    28      NA  14.9    66     5     6
##  6    23     299   8.6    65     5     7
##  7    19      99  13.8    59     5     8
##  8     8      19  20.1    61     5     9
##  9     7      NA   6.9    74     5    11
## 10    16     256   9.7    69     5    12
## # ... with 106 more rows
```

```
# aq %>% View()
```

```
# lm(Ozone ~ ., data = aq, na.action = na.fail)
```

```
complete.cases(aq)
```

```
##   [1]  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE
##  [13]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [25]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [37]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [49]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [61]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [73]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [85]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [97]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [109]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

```
na.omit(aq)
```

```
## # A tibble: 111 x 6
##    Ozone Solar.R  Wind  Temp Month   Day
##    <int>   <int> <dbl> <int> <int> <int>
##  1    41     190   7.4    67     5     1
```

```
##  2    36     118   8      72     5     2
##  3    12     149  12.6    74     5     3
##  4    18     313  11.5    62     5     4
##  5    23     299   8.6    65     5     7
##  6    19      99  13.8    59     5     8
##  7     8      19  20.1    61     5     9
##  8    16     256   9.7    69     5    12
##  9    11     290   9.2    66     5    13
## 10    14     274  10.9    68     5    14
## # ... with 101 more rows
```

```
# ?lm
```

```
# cbind(
#   aq,
#   predict(preProcess(aq, "bagImpute"), aq)
# ) %>% View()
```

```
set.seed(42)
fit_original = train(
  Ozone ~ ., data = aq,
  method = "lm",
  trControl = trainControl(method = "cv", number = 5),
  na.action = na.omit
)
fit_original$results
```

```
##   intercept    RMSE  Rsquared      MAE  RMSESD RsquaredSD    MAESD
## 1      TRUE 20.9273 0.6315529 15.94474 6.18771  0.1492251 4.319219
```

```
set.seed(42)
fit = train(
  Ozone ~ ., data = aq,
  method = "lm",
  trControl = trainControl(method = "cv", number = 5),
  preProcess = "knnImpute",
  na.action = na.pass
)
fit$results
```

```
new_obs = aq[1:3, ]
new_obs$Solar.R[1:2] = NA
new_obs
```

```
## # A tibble: 3 x 6
##   Ozone Solar.R  Wind  Temp Month   Day
##   <int>   <int> <dbl> <int> <int> <int>
## 1    41      NA   7.4    67     5     1
```

```
## 2    36      NA   8      72     5     2
## 3    12     149  12.6    74     5     3
```

```
# predict(fit, new_obs, na.action = na.pass)
```

```
predict(preProcess(aq, "medianImpute"), new_obs)
```

```
## # A tibble: 3 x 6
##   Ozone Solar.R  Wind  Temp Month   Day
##   <int>   <dbl> <dbl> <int> <int> <int>
## 1    41     207   7.4    67     5     1
## 2    36     207   8      72     5     2
## 3    12     149  12.6    74     5     3
```

```
library("tidyverse")
library("caret")
```

```
data("airquality")
airquality = as_tibble(airquality)
```

```
airquality
```

```
## # A tibble: 153 x 6
##    Ozone Solar.R  Wind  Temp Month   Day
##    <int>   <int> <dbl> <int> <int> <int>
##  1    41     190   7.4    67     5     1
##  2    36     118   8      72     5     2
##  3    12     149  12.6    74     5     3
##  4    18     313  11.5    62     5     4
##  5    NA      NA  14.3    56     5     5
##  6    28      NA  14.9    66     5     6
##  7    23     299   8.6    65     5     7
##  8    19      99  13.8    59     5     8
##  9     8      19  20.1    61     5     9
## 10    NA     194   8.6    69     5    10
## # ... with 143 more rows
```

```
aq = airquality %>%
  filter(!is.na(Ozone))
```

```
train(
  Ozone ~ .,
  data = aq,
  method = "lm",
  trControl = trainControl(method = "cv", number = 5),
  na.action = na.pass,
  preProcess = c("medianImpute")
```

```
)
```

```
## Linear Regression
##
## 116 samples
##   5 predictor
##
## Pre-processing: median imputation (5)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 93, 93, 93, 93, 92
## Resampling results:
##
##   RMSE      Rsquared   MAE
##   21.36886  0.6367566  15.81676
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

```
aq
```

```
## # A tibble: 116 x 6
##    Ozone Solar.R  Wind  Temp Month   Day
##    <int>   <int> <dbl> <int> <int> <int>
##  1    41     190   7.4    67     5     1
##  2    36     118   8      72     5     2
##  3    12     149  12.6    74     5     3
##  4    18     313  11.5    62     5     4
##  5    28      NA  14.9    66     5     6
##  6    23     299   8.6    65     5     7
##  7    19      99  13.8    59     5     8
##  8     8      19  20.1    61     5     9
##  9     7      NA   6.9    74     5    11
## 10    16     256   9.7    69     5    12
## # ... with 106 more rows
```

```
set.seed(42)
mod_fct = train(
  Ozone ~ as.factor(Month),
  data = aq,
  method = "lm",
  trControl = trainControl(method = "cv", number = 5)
)

set.seed(42)
mod_num = train(
  Ozone ~ Month,
  data = aq,
  method = "lm",
```

```
  trControl = trainControl(method = "cv", number = 5)
)
```

```
sqrt(var(aq$Ozone))
```

```
## [1] 32.98788
```

```
mod_num$finalModel
```

```
##
## Call:
## lm(formula = .outcome ~ ., data = dat)
##
## Coefficients:
## (Intercept)        Month
##      15.657        3.678
```

```
mod_fct$finalModel
```

```
##
## Call:
## lm(formula = .outcome ~ ., data = dat)
##
## Coefficients:
##         (Intercept)  `as.factor(Month)6`  `as.factor(Month)7`
##              23.615                5.829                35.500
## `as.factor(Month)8`  `as.factor(Month)9`
##              36.346                7.833
```

```
unique(aq$Month)
```

```
## [1] 5 6 7 8 9
```

```
# error due to lack of factor level
# predict(mod_fct, data.frame(Month = 12))
```

```
aq$Month[3:7] = NA
aq$Month = as.factor(aq$Month)
aq
```

```
## # A tibble: 116 x 6
##    Ozone Solar.R  Wind  Temp Month   Day
##    <int>   <int> <dbl> <int> <fct> <int>
##  1    41     190   7.4    67 5         1
##  2    36     118   8      72 5         2
##  3    12     149  12.6    74 <NA>      3
##  4    18     313  11.5    62 <NA>      4
##  5    28      NA  14.9    66 <NA>      6
##  6    23     299   8.6    65 <NA>      7
```

```
##  7    19      99  13.8    59 <NA>      8
##  8     8      19  20.1    61 5         9
##  9     7      NA   6.9    74 5        11
## 10    16     256   9.7    69 5        12
## # ... with 106 more rows
```

```
train(
  Ozone ~ .,
  data = aq,
  method = "lm",
  trControl = trainControl(method = "cv", number = 5),
  na.action = na.pass,
  preProcess = c("medianImpute")
)
```

```
## Linear Regression
##
## 116 samples
##   5 predictor
##
## Pre-processing: median imputation (8)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 93, 93, 93, 93, 92
## Resampling results:
##
##   RMSE      Rsquared   MAE
##   21.80517  0.6290986  16.29875
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

```
table(aq$Month)
```

```
##
##  5  6  7  8  9
## 21  9 26 26 29
```

```
predict(preProcess(predict(dummyVars(Ozone ~ ., data = aq), aq), method = "knnImpute"), predict(d
```

```
##         Solar.R        Wind        Temp    Month.5    Month.6     Month.7
## 1    0.05702761 -0.68871845 -1.14603406  2.0608504 -0.2957032 -0.55056939
## 2   -0.73285918 -0.52087950 -0.61891292  2.0608504 -0.2957032 -0.55056939
## 3   -0.39276904  0.76588578 -0.40806447  0.0274780  0.4336980 -0.55056939
## 4    1.40641756  0.45818104 -1.67315520  2.0608504 -0.2957032 -0.55056939
## 5    0.35542929  1.40926842 -1.25145829  0.5358211 -0.2957032 -0.08046783
## 6    1.25282846 -0.35304055 -1.35688252  1.5525073 -0.2957032 -0.55056939
## 7   -0.94130153  1.10156368 -1.98942789  1.0441642 -0.2957032 -0.55056939
## 8   -1.81895353  2.86387265 -1.77857943  2.0608504 -0.2957032 -0.55056939
## 9    0.44758275 -0.82858424 -0.40806447  2.0608504 -0.2957032 -0.55056939
```

```
## 10   0.78109051 -0.04533581 -0.93518561  2.0608504 -0.2957032 -0.55056939
## 11   1.15409261 -0.18520160 -1.25145829  2.0608504 -0.2957032 -0.55056939
## 12   0.97856221  0.29034209 -1.04060984  2.0608504 -0.2957032 -0.55056939
## 13  -1.31430363  0.93372473 -2.09485212  2.0608504 -0.2957032 -0.55056939
## 14   1.63680121  0.45818104 -1.46230675  2.0608504 -0.2957032 -0.55056939
## 15   1.34059366  0.59804683 -1.25145829  2.0608504 -0.2957032 -0.55056939
## 16  -1.17168518  2.38832896 -2.20027634  2.0608504 -0.2957032 -0.55056939
## 17   1.50515341  0.45818104 -1.04060984  2.0608504 -0.2957032 -0.55056939
## 18  -1.54468728 -0.04533581 -1.67315520  2.0608504 -0.2957032 -0.55056939
## 19  -1.93963068 -0.04533581 -1.98942789  2.0608504 -0.2957032 -0.55056939
## 20   1.48321211  1.88481211 -0.51348870  2.0608504 -0.2957032 -0.55056939
## 21  -1.75312963 -0.04533581 -1.77857943  2.0608504 -0.2957032 -0.55056939
## 22  -1.01809608  0.59804683 -1.77857943  2.0608504 -0.2957032 -0.55056939
## 23  -1.88477743  0.59804683 -1.14603406  2.0608504 -0.2957032 -0.55056939
## 24   0.73720791  1.40926842  0.32990513  2.0608504 -0.2957032 -0.55056939
## 25   0.41905906 -1.16426214  0.11905667  2.0608504 -0.2957032 -0.55056939
## 26   1.03341546 -0.68871845 -0.19721601  2.0608504 -0.2957032 -0.55056939
## 27  -0.63412333 -0.04533581  0.43532936 -0.4808651  3.3513029 -0.55056939
## 28   1.16506326  1.10156368  1.27872318 -0.4808651  3.3513029 -0.55056939
## 29   1.51612406  0.45818104  0.96245049 -0.4808651  3.3513029 -0.55056939
## 30  -0.40373969 -0.52087950  0.43532936 -0.4808651  3.3513029 -0.55056939
## 31   0.06799826  1.40926842 -0.09179178 -0.4808651  3.3513029 -0.55056939
## 32   1.08826871  3.03171160 -0.61891292 -0.4808651  3.3513029 -0.55056939
## 33  -1.62148183 -0.18520160 -1.35688252 -0.4808651  3.3513029 -0.55056939
## 34  -0.71091788  0.45818104 -0.51348870 -0.4808651  3.3513029 -0.55056939
## 35  -0.52441684  0.12250314 -0.19721601 -0.4808651  3.3513029 -0.55056939
## 36   0.92370896 -1.61183267  0.64617781 -0.4808651 -0.2957032  1.79993840
## 37   0.69332531 -0.18520160  0.75160204 -0.4808651 -0.2957032  1.79993840
## 38   0.56167751 -0.18520160  0.32990513 -0.4808651 -0.2957032  1.79993840
## 39  -0.10753214 -1.47196688  0.54075358 -0.4808651 -0.2957032  1.79993840
## 40   1.41738821  0.29034209  0.54075358 -0.4808651 -0.2957032  1.79993840
## 41   1.00050351 -1.33210109  1.06787472 -0.4808651 -0.2957032  1.79993840
## 42   0.90176766 -0.99642319  1.48957163 -0.4808651 -0.2957032  1.79993840
## 43   0.95662091 -1.16426214  1.48957163 -0.4808651 -0.2957032  1.79993840
## 44  -0.10753214 -0.68871845  1.17329895 -0.4808651 -0.2957032  1.79993840
## 45   0.86885571  1.24142947 -0.51348870 -0.4808651 -0.2957032  1.79993840
## 46  -0.10753214  1.40926842  0.32990513 -0.4808651 -0.2957032  1.79993840
## 47  -1.50080468  1.24142947  0.22448090 -0.4808651 -0.2957032  1.79993840
## 48   0.82497311 -0.82858424  0.32990513 -0.4808651 -0.2957032  1.79993840
## 49   0.97856221  0.12250314  0.43532936 -0.4808651 -0.2957032  1.79993840
## 50   1.09923936 -0.99642319  0.64617781 -0.4808651 -0.2957032  1.79993840
## 51   0.02411566 -1.33210109  0.96245049 -0.4808651 -0.2957032  1.79993840
## 52   0.38614711  0.45818104  0.75160204 -0.4808651 -0.2957032  1.79993840
## 53  -1.95060133 -0.82858424 -0.40806447 -0.4808651 -0.2957032  1.79993840
## 54   1.19797521 -0.35304055  0.85702627 -0.4808651 -0.2957032  1.79993840
## 55   0.41905906 -0.52087950  0.75160204 -0.4808651 -0.2957032  1.79993840
```

```
## 56  -1.13877323 -0.35304055  0.43532936 -0.4808651 -0.2957032  1.79993840
## 57  -1.12780258  0.59804683  0.85702627 -0.4808651 -0.2957032  1.79993840
## 58   0.30935256 -0.68871845  1.06787472 -0.4808651 -0.2957032  1.79993840
## 59   0.98953286 -0.68871845  0.85702627 -0.4808651 -0.2957032  1.79993840
## 60   0.74817856 -0.68871845  0.54075358 -0.4808651 -0.2957032  1.79993840
## 61   0.75914921 -0.18520160  0.32990513 -0.4808651 -0.2957032  1.79993840
## 62  -1.11683193 -0.82858424  0.32990513 -0.4808651 -0.2957032 -0.55056939
## 63  -1.76410028  1.10156368  0.32990513 -0.4808651 -0.2957032 -0.55056939
## 64  -1.18265583 -0.68871845  0.43532936 -0.4808651 -0.2957032 -0.55056939
## 65  -0.16019126 -0.82858424  0.85702627 -0.4808651 -0.2957032 -0.55056939
## 66  -0.16019126 -0.68871845  0.75160204 -0.4808651 -0.2957032 -0.55056939
## 67  -0.55074640 -1.47196688  0.96245049 -0.4808651 -0.2957032 -0.55056939
## 68   0.77011986 -1.63980583  1.17329895 -0.4808651 -0.2957032 -0.55056939
## 69   0.48488296  0.12250314  1.27872318 -0.4808651 -0.2957032 -0.55056939
## 70   0.24352866 -0.52087950  1.27872318 -0.4808651 -0.2957032 -0.55056939
## 71   0.07896891  0.45818104  0.85702627 -0.4808651 -0.2957032 -0.55056939
## 72   0.96759156  0.45818104  0.43532936 -0.4808651 -0.2957032 -0.55056939
## 73  -0.30500384 -0.04533581  0.22448090 -0.4808651 -0.2957032 -0.55056939
## 74  -1.24847973  0.12250314 -0.09179178 -0.4808651 -0.2957032 -0.55056939
## 75  -1.46789273 -0.99642319  0.11905667 -0.4808651 -0.2957032 -0.55056939
## 76  -0.76577113 -0.68871845 -0.19721601 -0.4808651 -0.2957032 -0.55056939
## 77   0.64944271  0.29034209  0.01363244 -0.4808651 -0.2957032 -0.55056939
## 78   0.05702761  0.12250314  0.01363244 -0.4808651 -0.2957032 -0.55056939
## 79   0.81400246  1.57710737 -0.09179178 -0.4808651 -0.2957032 -0.55056939
## 80  -1.63245248  1.24142947 -0.61891292 -0.4808651 -0.2957032 -0.55056939
## 81   0.29838191 -0.04533581  0.11905667 -0.4808651 -0.2957032 -0.55056939
## 82   0.58361881 -1.80764478  0.32990513 -0.4808651 -0.2957032 -0.55056939
## 83   0.33129386 -0.52087950  0.85702627 -0.4808651 -0.2957032 -0.55056939
## 84   0.19964606 -0.04533581  2.01669277 -0.4808651 -0.2957032 -0.55056939
## 85   0.44100036 -2.11534952  1.70042009 -0.4808651 -0.2957032 -0.55056939
## 86   0.57264816 -0.99642319  1.91126855 -0.4808651 -0.2957032 -0.55056939
## 87   0.03508631 -0.99642319  1.70042009 -0.4808651 -0.2957032 -0.55056939
## 88  -0.19529734 -0.82858424  1.38414741 -0.4808651 -0.2957032 -0.55056939
## 89   0.13382216 -1.33210109  1.48957163 -0.4808651 -0.2957032 -0.55056939
## 90  -0.01976694 -1.97548373  1.59499586 -0.4808651 -0.2957032 -0.55056939
## 91   0.04605696 -1.47196688  1.59499586 -0.4808651 -0.2957032 -0.55056939
## 92  -0.98518413 -0.68871845  0.96245049 -0.4808651 -0.2957032 -0.55056939
## 93  -1.01809608  1.57710737  0.64617781 -0.4808651 -0.2957032 -0.55056939
## 94   0.73720791  0.29034209  0.22448090 -0.4808651 -0.2957032 -0.55056939
## 95   0.38614711  0.12250314  0.01363244 -0.4808651 -0.2957032 -0.55056939
## 96   0.49585361  0.29034209 -0.30264024 -0.4808651 -0.2957032 -0.55056939
## 97   0.81400246 -0.04533581 -0.51348870 -0.4808651 -0.2957032 -0.55056939
## 98   0.56167751  1.40926842  0.32990513 -0.4808651 -0.2957032 -0.55056939
## 99   0.81400246  1.57710737 -0.19721601 -0.4808651 -0.2957032 -0.55056939
## 100  0.58361881 -0.99642319 -0.09179178 -0.4808651 -0.2957032 -0.55056939
## 101 -1.76410028  0.29034209 -0.72433715 -0.4808651 -0.2957032 -0.55056939
```

```
## 102 -0.79868308  0.45818104 -0.72433715 -0.4808651 -0.2957032 -0.55056939
## 103  0.57264816 -0.82858424  0.01363244 -0.4808651 -0.2957032 -0.55056939
## 104  0.43002971  1.10156368 -1.14603406 -0.4808651 -0.2957032 -0.55056939
## 105 -1.73118833  0.12250314 -0.19721601 -0.4808651 -0.2957032 -0.55056939
## 106  0.58361881  0.12250314 -1.04060984 -0.4808651 -0.2957032 -0.55056939
## 107  0.17770476 -0.52087950  0.43532936 -0.4808651 -0.2957032 -0.55056939
## 108  0.58361881  0.76588578 -1.46230675 -0.4808651 -0.2957032 -0.55056939
## 109 -1.87380678 -0.18520160 -0.72433715 -0.4808651 -0.2957032 -0.55056939
## 110 -0.50247554  0.12250314  0.32990513 -0.4808651 -0.2957032 -0.55056939
## 111 -1.48983403  0.12250314 -0.93518561 -0.4808651 -0.2957032 -0.55056939
## 112 -1.80798288  1.88481211 -1.56773098 -0.4808651 -0.2957032 -0.55056939
## 113  0.08993956 -0.82858424 -0.82976138 -0.4808651 -0.2957032 -0.55056939
## 114  0.06799826  1.24142947 -0.30264024 -0.4808651 -0.2957032 -0.55056939
## 115 -0.59024074 -0.52087950 -0.19721601 -0.4808651 -0.2957032 -0.55056939
## 116  0.41905906  0.45818104 -1.04060984 -0.4808651 -0.2957032 -0.55056939
##        Month.8    Month.9         Day
## 1   -0.5505694 -0.5920071 -1.66106728
## 2   -0.5505694 -0.5920071 -1.54678270
## 3   -0.5505694  0.7675678 -1.43249811
## 4   -0.5505694 -0.5920071 -1.31821353
## 5   -0.5505694  0.3143762 -1.08964437
## 6   -0.5505694 -0.1388155 -0.97535979
## 7   -0.5505694  0.3143762 -0.86107521
## 8   -0.5505694 -0.5920071 -0.74679063
## 9   -0.5505694 -0.5920071 -0.51822146
## 10  -0.5505694 -0.5920071 -0.40393688
## 11  -0.5505694 -0.5920071 -0.28965230
## 12  -0.5505694 -0.5920071 -0.17536772
## 13  -0.5505694 -0.5920071 -0.06108314
## 14  -0.5505694 -0.5920071  0.05320144
## 15  -0.5505694 -0.5920071  0.16748602
## 16  -0.5505694 -0.5920071  0.28177061
## 17  -0.5505694 -0.5920071  0.39605519
## 18  -0.5505694 -0.5920071  0.51033977
## 19  -0.5505694 -0.5920071  0.62462435
## 20  -0.5505694 -0.5920071  0.73890893
## 21  -0.5505694 -0.5920071  0.85319351
## 22  -0.5505694 -0.5920071  0.96747809
## 23  -0.5505694 -0.5920071  1.42461642
## 24  -0.5505694 -0.5920071  1.53890100
## 25  -0.5505694 -0.5920071  1.65318558
## 26  -0.5505694 -0.5920071  1.76747016
## 27  -0.5505694 -0.5920071 -0.97535979
## 28  -0.5505694 -0.5920071 -0.74679063
## 29  -0.5505694 -0.5920071 -0.63250605
## 30  -0.5505694 -0.5920071 -0.28965230
```

```
## 31  -0.5505694 -0.5920071  0.05320144
## 32  -0.5505694 -0.5920071  0.16748602
## 33  -0.5505694 -0.5920071  0.28177061
## 34  -0.5505694 -0.5920071  0.39605519
## 35  -0.5505694 -0.5920071  0.51033977
## 36  -0.5505694 -0.5920071 -1.66106728
## 37  -0.5505694 -0.5920071 -1.54678270
## 38  -0.5505694 -0.5920071 -1.43249811
## 39  -0.5505694 -0.5920071 -1.20392895
## 40  -0.5505694 -0.5920071 -1.08964437
## 41  -0.5505694 -0.5920071 -0.97535979
## 42  -0.5505694 -0.5920071 -0.86107521
## 43  -0.5505694 -0.5920071 -0.74679063
## 44  -0.5505694 -0.5920071 -0.63250605
## 45  -0.5505694 -0.5920071 -0.40393688
## 46  -0.5505694 -0.5920071 -0.28965230
## 47  -0.5505694 -0.5920071 -0.06108314
## 48  -0.5505694 -0.5920071  0.05320144
## 49  -0.5505694 -0.5920071  0.16748602
## 50  -0.5505694 -0.5920071  0.28177061
## 51  -0.5505694 -0.5920071  0.39605519
## 52  -0.5505694 -0.5920071  0.51033977
## 53  -0.5505694 -0.5920071  0.62462435
## 54  -0.5505694 -0.5920071  0.96747809
## 55  -0.5505694 -0.5920071  1.08176268
## 56  -0.5505694 -0.5920071  1.19604726
## 57  -0.5505694 -0.5920071  1.31033184
## 58  -0.5505694 -0.5920071  1.42461642
## 59  -0.5505694 -0.5920071  1.53890100
## 60  -0.5505694 -0.5920071  1.65318558
## 61  -0.5505694 -0.5920071  1.76747016
## 62   1.7999384 -0.5920071 -1.66106728
## 63   1.7999384 -0.5920071 -1.54678270
## 64   1.7999384 -0.5920071 -1.43249811
## 65   1.7999384 -0.5920071 -1.31821353
## 66   1.7999384 -0.5920071 -1.20392895
## 67   1.7999384 -0.5920071 -1.08964437
## 68   1.7999384 -0.5920071 -0.97535979
## 69   1.7999384 -0.5920071 -0.86107521
## 70   1.7999384 -0.5920071 -0.74679063
## 71   1.7999384 -0.5920071 -0.40393688
## 72   1.7999384 -0.5920071 -0.28965230
## 73   1.7999384 -0.5920071 -0.17536772
## 74   1.7999384 -0.5920071  0.05320144
## 75   1.7999384 -0.5920071  0.16748602
## 76   1.7999384 -0.5920071  0.28177061
```

```
## 77   1.7999384 -0.5920071  0.39605519
## 78   1.7999384 -0.5920071  0.51033977
## 79   1.7999384 -0.5920071  0.62462435
## 80   1.7999384 -0.5920071  0.73890893
## 81   1.7999384 -0.5920071  0.96747809
## 82   1.7999384 -0.5920071  1.08176268
## 83   1.7999384 -0.5920071  1.19604726
## 84   1.7999384 -0.5920071  1.42461642
## 85   1.7999384 -0.5920071  1.53890100
## 86   1.7999384 -0.5920071  1.65318558
## 87   1.7999384 -0.5920071  1.76747016
## 88  -0.5505694  1.6739512 -1.66106728
## 89  -0.5505694  1.6739512 -1.54678270
## 90  -0.5505694  1.6739512 -1.43249811
## 91  -0.5505694  1.6739512 -1.31821353
## 92  -0.5505694  1.6739512 -1.20392895
## 93  -0.5505694  1.6739512 -1.08964437
## 94  -0.5505694  1.6739512 -0.97535979
## 95  -0.5505694  1.6739512 -0.86107521
## 96  -0.5505694  1.6739512 -0.74679063
## 97  -0.5505694  1.6739512 -0.63250605
## 98  -0.5505694  1.6739512 -0.51822146
## 99  -0.5505694  1.6739512 -0.40393688
## 100 -0.5505694  1.6739512 -0.28965230
## 101 -0.5505694  1.6739512 -0.17536772
## 102 -0.5505694  1.6739512 -0.06108314
## 103 -0.5505694  1.6739512  0.05320144
## 104 -0.5505694  1.6739512  0.16748602
## 105 -0.5505694  1.6739512  0.28177061
## 106 -0.5505694  1.6739512  0.39605519
## 107 -0.5505694  1.6739512  0.51033977
## 108 -0.5505694  1.6739512  0.62462435
## 109 -0.5505694  1.6739512  0.73890893
## 110 -0.5505694  1.6739512  0.85319351
## 111 -0.5505694  1.6739512  0.96747809
## 112 -0.5505694  1.6739512  1.08176268
## 113 -0.5505694  1.6739512  1.19604726
## 114 -0.5505694  1.6739512  1.42461642
## 115 -0.5505694  1.6739512  1.53890100
## 116 -0.5505694  1.6739512  1.65318558
```

```
predict(dummyVars(Ozone ~ ., data = aq), aq)
```

```
##     Solar.R Wind Temp Month.5 Month.6 Month.7 Month.8 Month.9 Day
## 1       190  7.4   67       1       0       0       0       0   1
## 2       118  8.0   72       1       0       0       0       0   2
```

```
## 3       149 12.6   74       NA       NA       NA       NA       NA   3
## 4       313 11.5   62       NA       NA       NA       NA       NA   4
## 5        NA 14.9   66       NA       NA       NA       NA       NA   6
## 6       299  8.6   65       NA       NA       NA       NA       NA   7
## 7        99 13.8   59       NA       NA       NA       NA       NA   8
## 8        19 20.1   61        1        0        0        0        0   9
## 9        NA  6.9   74        1        0        0        0        0  11
## 10      256  9.7   69        1        0        0        0        0  12
## 11      290  9.2   66        1        0        0        0        0  13
## 12      274 10.9   68        1        0        0        0        0  14
## 13       65 13.2   58        1        0        0        0        0  15
## 14      334 11.5   64        1        0        0        0        0  16
## 15      307 12.0   66        1        0        0        0        0  17
## 16       78 18.4   57        1        0        0        0        0  18
## 17      322 11.5   68        1        0        0        0        0  19
## 18       44  9.7   62        1        0        0        0        0  20
## 19        8  9.7   59        1        0        0        0        0  21
## 20      320 16.6   73        1        0        0        0        0  22
## 21       25  9.7   61        1        0        0        0        0  23
## 22       92 12.0   61        1        0        0        0        0  24
## 23       13 12.0   67        1        0        0        0        0  28
## 24      252 14.9   81        1        0        0        0        0  29
## 25      223  5.7   79        1        0        0        0        0  30
## 26      279  7.4   76        1        0        0        0        0  31
## 27      127  9.7   82        0        1        0        0        0   7
## 28      291 13.8   90        0        1        0        0        0   9
## 29      323 11.5   87        0        1        0        0        0  10
## 30      148  8.0   82        0        1        0        0        0  13
## 31      191 14.9   77        0        1        0        0        0  16
## 32      284 20.7   72        0        1        0        0        0  17
## 33       37  9.2   65        0        1        0        0        0  18
## 34      120 11.5   73        0        1        0        0        0  19
## 35      137 10.3   76        0        1        0        0        0  20
## 36      269  4.1   84        0        0        1        0        0   1
## 37      248  9.2   85        0        0        1        0        0   2
## 38      236  9.2   81        0        0        1        0        0   3
## 39      175  4.6   83        0        0        1        0        0   5
## 40      314 10.9   83        0        0        1        0        0   6
## 41      276  5.1   88        0        0        1        0        0   7
## 42      267  6.3   92        0        0        1        0        0   8
## 43      272  5.7   92        0        0        1        0        0   9
## 44      175  7.4   89        0        0        1        0        0  10
## 45      264 14.3   73        0        0        1        0        0  12
## 46      175 14.9   81        0        0        1        0        0  13
## 47       48 14.3   80        0        0        1        0        0  15
## 48      260  6.9   81        0        0        1        0        0  16
```

```
## 49  274 10.3  82    0    0    1    0    0  17
## 50  285  6.3  84    0    0    1    0    0  18
## 51  187  5.1  87    0    0    1    0    0  19
## 52  220 11.5  85    0    0    1    0    0  20
## 53    7  6.9  74    0    0    1    0    0  21
## 54  294  8.6  86    0    0    1    0    0  24
## 55  223  8.0  85    0    0    1    0    0  25
## 56   81  8.6  82    0    0    1    0    0  26
## 57   82 12.0  86    0    0    1    0    0  27
## 58  213  7.4  88    0    0    1    0    0  28
## 59  275  7.4  86    0    0    1    0    0  29
## 60  253  7.4  83    0    0    1    0    0  30
## 61  254  9.2  81    0    0    1    0    0  31
## 62   83  6.9  81    0    0    0    1    0   1
## 63   24 13.8  81    0    0    0    1    0   2
## 64   77  7.4  82    0    0    0    1    0   3
## 65   NA  6.9  86    0    0    0    1    0   4
## 66   NA  7.4  85    0    0    0    1    0   5
## 67   NA  4.6  87    0    0    0    1    0   6
## 68  255  4.0  89    0    0    0    1    0   7
## 69  229 10.3  90    0    0    0    1    0   8
## 70  207  8.0  90    0    0    0    1    0   9
## 71  192 11.5  86    0    0    0    1    0  12
## 72  273 11.5  82    0    0    0    1    0  13
## 73  157  9.7  80    0    0    0    1    0  14
## 74   71 10.3  77    0    0    0    1    0  16
## 75   51  6.3  79    0    0    0    1    0  17
## 76  115  7.4  76    0    0    0    1    0  18
## 77  244 10.9  78    0    0    0    1    0  19
## 78  190 10.3  78    0    0    0    1    0  20
## 79  259 15.5  77    0    0    0    1    0  21
## 80   36 14.3  72    0    0    0    1    0  22
## 81  212  9.7  79    0    0    0    1    0  24
## 82  238  3.4  81    0    0    0    1    0  25
## 83  215  8.0  86    0    0    0    1    0  26
## 84  203  9.7  97    0    0    0    1    0  28
## 85  225  2.3  94    0    0    0    1    0  29
## 86  237  6.3  96    0    0    0    1    0  30
## 87  188  6.3  94    0    0    0    1    0  31
## 88  167  6.9  91    0    0    0    0    1   1
## 89  197  5.1  92    0    0    0    0    1   2
## 90  183  2.8  93    0    0    0    0    1   3
## 91  189  4.6  93    0    0    0    0    1   4
## 92   95  7.4  87    0    0    0    0    1   5
## 93   92 15.5  84    0    0    0    0    1   6
## 94  252 10.9  80    0    0    0    0    1   7
```

```
## 95      220 10.3   78       0       0       0       0       1   8
## 96      230 10.9   75       0       0       0       0       1   9
## 97      259  9.7   73       0       0       0       0       1  10
## 98      236 14.9   81       0       0       0       0       1  11
## 99      259 15.5   76       0       0       0       0       1  12
## 100     238  6.3   77       0       0       0       0       1  13
## 101      24 10.9   71       0       0       0       0       1  14
## 102     112 11.5   71       0       0       0       0       1  15
## 103     237  6.9   78       0       0       0       0       1  16
## 104     224 13.8   67       0       0       0       0       1  17
## 105      27 10.3   76       0       0       0       0       1  18
## 106     238 10.3   68       0       0       0       0       1  19
## 107     201  8.0   82       0       0       0       0       1  20
## 108     238 12.6   64       0       0       0       0       1  21
## 109      14  9.2   71       0       0       0       0       1  22
## 110     139 10.3   81       0       0       0       0       1  23
## 111      49 10.3   69       0       0       0       0       1  24
## 112      20 16.6   63       0       0       0       0       1  25
## 113     193  6.9   70       0       0       0       0       1  26
## 114     191 14.3   75       0       0       0       0       1  28
## 115     131  8.0   76       0       0       0       0       1  29
## 116     223 11.5   68       0       0       0       0       1  30
```

```
data("airquality")
airquality$Month[airquality$Month == 5] = NA
airquality$Month = factor(airquality$Month)
airquality$Month
```

```
##   [1] <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA>
##  [16] <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA>
##  [31] <NA> 6    6    6    6    6    6    6    6    6    6    6    6    6    6
##  [46] 6    6    6    6    6    6    6    6    6    6    6    6    6    6    6
##  [61] 6    7    7    7    7    7    7    7    7    7    7    7    7    7    7
##  [76] 7    7    7    7    7    7    7    7    7    7    7    7    7    7    7
##  [91] 7    7    8    8    8    8    8    8    8    8    8    8    8    8    8
## [106] 8    8    8    8    8    8    8    8    8    8    8    8    8    8    8
## [121] 8    8    8    9    9    9    9    9    9    9    9    9    9    9    9
## [136] 9    9    9    9    9    9    9    9    9    9    9    9    9    9    9
## [151] 9    9    9
## Levels: 6 7 8 9
```

```
fct_explicit_na(airquality$Month, na_level = "(Missing)")
```

```
##   [1] (Missing) (Missing) (Missing) (Missing) (Missing) (Missing) (Missing)
##   [8] (Missing) (Missing) (Missing) (Missing) (Missing) (Missing) (Missing)
##  [15] (Missing) (Missing) (Missing) (Missing) (Missing) (Missing) (Missing)
##  [22] (Missing) (Missing) (Missing) (Missing) (Missing) (Missing) (Missing)
```

```
##  [29] (Missing) (Missing) (Missing) 6         6         6         6
##  [36] 6         6         6         6         6         6         6
##  [43] 6         6         6         6         6         6         6
##  [50] 6         6         6         6         6         6         6
##  [57] 6         6         6         6         6         7         7
##  [64] 7         7         7         7         7         7         7
##  [71] 7         7         7         7         7         7         7
##  [78] 7         7         7         7         7         7         7
##  [85] 7         7         7         7         7         7         7
##  [92] 7         8         8         8         8         8         8
##  [99] 8         8         8         8         8         8         8
## [106] 8         8         8         8         8         8         8
## [113] 8         8         8         8         8         8         8
## [120] 8         8         8         8         9         9         9
## [127] 9         9         9         9         9         9         9
## [134] 9         9         9         9         9         9         9
## [141] 9         9         9         9         9         9         9
## [148] 9         9         9         9         9         9
## Levels: 6 7 8 9 (Missing)
```

```
head(aq, n = 10)
```

```
## # A tibble: 10 x 6
##    Ozone Solar.R  Wind  Temp Month   Day
##    <int>   <int> <dbl> <int> <fct> <int>
##  1    41     190   7.4    67 5         1
##  2    36     118   8      72 5         2
##  3    12     149  12.6    74 <NA>      3
##  4    18     313  11.5    62 <NA>      4
##  5    28      NA  14.9    66 <NA>      6
##  6    23     299   8.6    65 <NA>      7
##  7    19      99  13.8    59 <NA>      8
##  8     8      19  20.1    61 5         9
##  9     7      NA   6.9    74 5        11
## 10    16     256   9.7    69 5        12
```

```
library("randomForest")
```

```
## randomForest 4.6-14
```

```
## Type rfNews() to see new features/changes/bug fixes.
```

```
##
## Attaching package: 'randomForest'
```

```
## The following object is masked from 'package:dplyr':
##
##     combine
```

```
## The following object is masked from 'package:ggplot2':
##
##     margin
```

```
randomForest(Ozone ~ ., data = rfImpute(Ozone ~ ., data = aq))
```

```
##      |      Out-of-bag   |
## Tree |      MSE  %Var(y) |
##  300 |    341.6    31.67 |
##      |      Out-of-bag   |
## Tree |      MSE  %Var(y) |
##  300 |    349.1    32.36 |
##      |      Out-of-bag   |
## Tree |      MSE  %Var(y) |
##  300 |    349.1    32.36 |
##      |      Out-of-bag   |
## Tree |      MSE  %Var(y) |
##  300 |    349.6    32.41 |
##      |      Out-of-bag   |
## Tree |      MSE  %Var(y) |
##  300 |    342.7    31.76 |
##      |      Out-of-bag   |
## Tree |      MSE  %Var(y) |
##  300 |    342.8    31.78 |
##      |      Out-of-bag   |
## Tree |      MSE  %Var(y) |
##  300 |    347.5    32.21 |
##      |      Out-of-bag   |
## Tree |      MSE  %Var(y) |
##  300 |    339.9    31.51 |
##      |      Out-of-bag   |
## Tree |      MSE  %Var(y) |
##  300 |    337.8    31.31 |
##      |      Out-of-bag   |
## Tree |      MSE  %Var(y) |
##  300 |    347.1    32.18 |

##
## Call:
##  randomForest(formula = Ozone ~ ., data = rfImpute(Ozone ~ .,      data = aq))
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 1
##
##           Mean of squared residuals: 340.1159
##                     % Var explained: 68.47
```

```
head(na.roughfix(aq), n = 10)
```

```
## # A tibble: 10 x 6
##    Ozone Solar.R  Wind  Temp Month   Day
##    <int>   <int> <dbl> <int> <fct> <int>
##  1    41     190   7.4    67 5         1
##  2    36     118   8      72 5         2
##  3    12     149  12.6    74 9         3
##  4    18     313  11.5    62 9         4
##  5    28     207  14.9    66 9         6
##  6    23     299   8.6    65 9         7
##  7    19      99  13.8    59 9         8
##  8     8      19  20.1    61 5         9
##  9     7     207   6.9    74 5        11
## 10    16     256   9.7    69 5        12
```